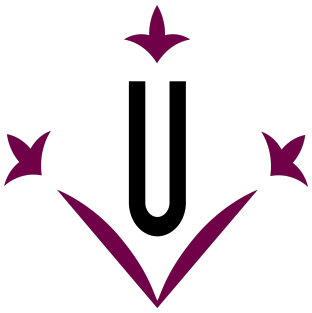

UNIVERSIAT DE LLEIDA

Escola Politècnica Superior

Programació amb OpenSSL en Python

Alejandro Contreras Montañez

Director: César Fernández Camon

Treball Final de Grau, Juny 2016

Índex

Índex de figures

Índex de taules

Capítol 1

Introducció

Un punt important en les aplicacions és la seguretat ja que avui en dia es mou un gran volum d'informació, per tant la criptografia és una de les principals eines per aconseguir proveir aquesta seguretat. Els objectius principals són la confidencialitat, la integritat, l'autenticació i el no repudi de les dades.
OpenSSL és una llibreria criptogràfica que ens proveeix implementacions dels millors algorismes actuals, incloent algorismes d'encriptació com 3DES, RSA i AES, algorismes de hash i Codis d'Autentificació de Missatges (MAC). També permeten crear certificats digitals. Amb OpenSSL podem implementar protocols de seguretat com Secure Socket Layer (SSL) i Transport Socket Layer (TLS). L'objectiu d'aquest treball és disposar d'eines i exemples de desenvolupament sobre OpenSSL des de llenguatges d'alt nivell com python, evitant l'ús del llenguatge C, per tal de facilitar la comprensió dels alumnes evitant tenir que inicialitzar i assignar blocs de memòria a les variables i evitar inicialitzar el context abans de xifrar o desxifrar. La implementació d'aquestes funcions la realitzarem mitjançant la llibreria M2Crypto de Matej Cepl [1] [18]. M2Crypto és el wrapper de python per a OpenSSL més complet, ja que ens proporciona gran part de les seves funcionalitats com MAC, RSA, DSA, DH, SSL, HTTPS i S/MIME. Tots els scripts en python d'aquest treball han estat implementats utilitzant python 2.7.6.

Capítol 2

Operacions per al xifrat simètric

2.1 BIO

La llibreria M2Crypto ens proporciona el mòdul BIO i la classe BIO per tal de poder llegir i escriure en un buffer similar a un fitxer.
Els mètodes més rellevants que ens proporciona la classe BIO són:

readable(self)

Comprova si es pot llegir al buffer.

writeable(self)

Comprova si es pot escriure al buffer.

readline(self, size=4096)

Llegeix n bytes del buffer o fins trobar un salt de línia.

size (int)

Nombre de bytes a llegir.

write(self, data)

Escriu data al buffer.

data (str)

Dades a escriure.

El mòdul BIO també ens proporciona la classe MemoryBuffer que estén la classe BIO. Ens proporciona els següents mètodes:

read(self, size=0)

Llegeix size bytes del buffer.

size (int)

Nombre de bytes a llegir.

write_close(self)

Tanca l'escriptura al buffer.

2.2 Generador aleatori

Una part important de la seguretat en SSL és la generació de claus aleatòries. Per tant, hem de poder utilitzar una llavor (seed) amb suficient entropia, és a dir, major que la longitud de la clau. En aquest cas podem definir com entropia el nombre de bits aleatoris desconeguts.

$$B_{entropy} = \sum_{i=1}^{n} (i - \log_2(p_i)) \tag{2.1}$$

on:

i: Espai possibles valors

p_i: Probabilitat d'i

La llibreria M2Crypto disposa del mòdul Rand que utilitza el PRNG (Pseudo Random Number Generator) d'OpenSSL. Les funcions de les quals disposem són les següents:

rand_seed(buf, num)

Barreja num bytes de buf al PRNG si num és igual a l'entropia.

buf

Bytes a barrejar.

num (int)

Nombre de bytes a barrejar.

rand_add(buf, num)

Barreja num bytes de buf al PRNG.

buf

Bytes a barrejar.

num (int)

Nombre de bytes a barrejar.

load_file(filename, max_bytes)

Llegeix max_bytes bytes del fitxer filename i els afegeix al PRNG.

filename (str)

Fitxer d'on llegir els bytes.

max_bytes (long)

Màxim nombre de bytes a llegir.

save_file(filename)

Escriu 1024 bytes aleatoris al fitxer filename.

filename (str)

Fitxer on escriure.

rand_pseudo_bytes(num)

Retorna una tupla amb num bytes aleatoris i un enter que val 1 si els bytes generats són criptogràficament forts o 0 en el cas contrari. Aquesta funció està obsoleta tot i que pot ser utilitzada amb propòsits no criptogràfics.

num (int)

Nombre de bytes a generar.

La llibreria M2Crypto també ens proporciona el mòdul BN per tal de generar nombres aleatoris. Les funcions de les quals disposem són:

rand_range(range)

Retorna un número aleatori entre 0 i range (0 ¡= rand ¡ range).

range (long)

Límit superior del número aleatori a generar.

rand(bits, top, bottom)

Retorna un número, de tipus long, aleatori de n bits.

bits (int)

Nombre de bits que contindrà el número a generar.

top (int)

Si top val -1 el bit més significatiu pot valdre tant 0 com 1, si val 0 el bit més significatiu valdrà 1 i si val 1 els dos bits més significatius valdran 1.

bottom (int)

Si bottom val 0 el bit menys significatiu pot valdre tant 0 com 1, mentre que si val 1, el bit menys significatiu valdrà 1, per tant, el nombre serà imparell.

randfname(bits)

Retorna nom de fitxer aleatori de n bits.

bits (int)

Nombre de bits que contindrà el nom a generar.

En aquest cas els nombres aleatoris generats seran del tipus long ja que amb python podem operar amb grans nombres utilitzant aquest tipus i per tant, nos ens farà falta utilitzar BIGNUMs.

Finalment en el mòdul util disposem de la següent funció per passar de string de bytes a número.

octx_to_num(x)

Retorna la representació numèrica de x.

x (str)

String de bytes.

2.3 Xifrat simètric

El xifrat simètric o xifrat de clau secreta és un mètode criptogràfic en el qual s'utilitza una clau per xifrar i desxifrar missatges entre l'emissor i el receptor, per tant, les dos parts han d'acordar quina serà la clau compartida.
Aquest tipus de xifrat basa tota la seguretat en la clau i no l'algorisme, ja que, tot i saber el funcionament de l'algorisme sense la clau no podem desxifrar el missatge. Per tant, quan més gran és l'espai de claus més segur és l'algorisme.
El major problema d'aquest tipus de xifrat és l'intercanvi de claus ja que per poder intercanviar les claus l'emissor i el receptor han d'utilitzar un canal segur per intercanviar-la, sinó, no podem garantir que la clau sigui segura i no hagi arribat a tercers. Un altre problema és que si tenim n persones que es volen

comunicar entre elles, necessitem una clau per cada parella de persones, és a dir necessitaríem n(n-1)/2 claus.
Per tal d'obtenir una clau més segura podem utilitzar la funció de derivació de claus Password-Based Key Derivation Function (PBKDF) [9]. La qual concatena la nostra clau amb un salt i aplica una funció de hash, habitualment MD5, sobre el resultat. Després torna a aplicar una funció de hash sobre el hash anterior i això es repeteix durant el nombre de cicles indicat.
Exemple de funcionament:

Password (P), Salt (S), Nombre_Cicles (C)

- T1 = HASH (P||S)
- T2 = HASH (T1)
- ...
- Key = HASH (TC)

Utilitzant aquesta funció aconseguim aleatoritzar la generació de claus i dificultem l'atac per diccionari.

La llibreria M2Crypto ens proporciona la funció pbkdf2 del mòdul EVP per tal de poder generar una clau utilitzant la funció pbkdf.

pbkdf2(password, salt, iter, keylen)

Retorna una string amb la clau derivada.
password (str)
Contrasenya a partir de la qual derivarem la clau.
salt (str)
Bytes que concatenarem a password per tal de derivar la clau.
iter(int)
Nombre de cicles a dur a terme.
keylen (int)
Llargada de la clau en bytes.

Dins del mòdul EVP també trobem la classe Cipher que ens permet xifrar i desxifrar mitjançant xifrat simètric. L'especificació del constructor és la següent:

Cipher (self, alg, key, iv, op, key_as_bytes=0, d='md5', salt='12345678', i=1, padding=1)

alg (str)

Algorisme a utilitzar a l'hora de xifrar o desxifrar.

key (str)

Clau a utilitzar per xifrar o desxifrar.

iv (str)

Vector d'inicialització en cas de que sigui necessari.

op (int)

Operació a dur a terme. 0 si volem desxifrar i 1 si volem xifrar.

key_as_bytes (int)

0 si no volem aplicar funció de derivació de clau i 1 si volem. Per defecte val 0.

d (str)

Algorisme de hash a utilitzar en la derivació de clau. Per defecte utilitza md5.

salt (str)

Salt a utilitzar en la derivació de clau. Per defecte val '12345678'.

i (int)

Nombre d'iteracions a aplicar en la derivació de clau. Per defecte val 1.

padding (int)

0 si no volem afegir padding a cada bloc i 1 si volem afegir. Per defecte val 1.

Els mètodes que ens proporciona són:

update(self, data)

Retorna data xifrat

data (str)

Dades a xifrar

final(self)

Retorna el darrer bloc xifrat en cas d'utilitzar padding. També comprova que el xifrat o desxifrat sigui correcte, en cas contrari retorna un error indicant que és incorrecte.

set_padding(self, padding=1)

Estableix si es vol utilitzar padding.

padding (int)

0 si no volem afegir padding i 1 si volem. Per defecte val 1.

2.3.1 Exemple xifrat simètric

Donat un missatge encriptat en DES (des_ede_cbc), sabent que la clau està en format pbkdf amb una mida de 256 bits i el nombre d'iteracions és 1, desxifrar el missatge emprant un diccionari de claus.

```
from M2Crypto.EVP import Cipher, pbkdf2,

def decrypt_message(enc_data, password):
    passwd = pbkdf2(password, "Salted__", 1, 32)
    c = Cipher(alg='des_ede_cbc', key=passwd, iv="", \
     op=0, padding=1)
    message = c.update(enc_data)
    try:
        c.final()
        message.decode('ascii')
        return message
    except:
        return ""


if __name__ == "__main__":
    dictionary = open("dictionary.txt", 'r')
    enc_file = open("encrypted.bin", 'r')
    enc_data = enc_file.read()
    for line in dictionary.readlines():
        passwd = line.strip()
        message = decrypt_message(enc_data, passwd)
        if message != "":
            break
    print "Password: " + passwd
    print "Message: " + message
    dictionary.close()
    enc_file.close()
```

Listing 2.1: Xifrat Simètric

2.4 Message digest

Els message digests, també coneguts com funcions de hash, són funcions que reben un missatge de longitud variable i generen un resum de longitud fixa depenent de la funció que s'utilitza. Aquest hash que generen serveix per garantir la integritat del missatge.
Exemple de funcionament:

Funció de hash (F), Missatge (M), Resum (H)

H = F(M)

El mòdul EVP ens proporciona la classe MessageDigest per tal de poder utilitzar les funcions hash. L'especificació del seu constructor és la següent:

MessageDigest(self, algo)

algo (str)

Funció de hash a aplicar.

Els mètodes que ens proporciona són els següents:

update(self, data)

Afegeix les dades a les quals s'aplicarà la funció de hash.

data (str)

Dades a les quals s'aplicarà la funció de hash.

final(self)

Aplica la funció de hash sobre les dades.

2.5 HMAC

Hash-based Message Authentication Code (HMAC) [4] és un tipus de codi d'autentificació de missatge el qual utilitza una funció hash, habitualment md5 o sha-1, i una clau per tal de poder verificar la integritat i autentificació de les dades. La seva fortalesa depèn de la funció hash, el nombre de bits que retorni aquesta funció i de la mida i aleatorietat de la clau.
Exemple de funcionament:

Clau (C), Missatge (M), byte 0x36 repetit N vegades (ipad), byte 0x5C repetit N vegades (opad)

HMAC(C, M) = HASH(C xor opad||HASH(C xor ipad)||M)

El mòdul EVP ens proporciona la classe HMAC per tal de dur a terme això. L'especificació del seu constructor és la següent:

HMAC(self, key, algo='sha1')

key (str)

Clau per dur a terme la HMAC.

algo (str)

Funció de hash a utilitzar. Per defecte utilitza sha1.

Aquesta classe ens proporciona els següents mètodes:

reset(self, key)

Canvia la clau introduïda prèviament per key.

key (str)

Nova clau a utilitzar.

update(self, data)

Afegeix les dades a les quals s'aplicarà la funció d'HMAC.

data (str)

Dades a les quals s'aplicarà la funció d'HMAC.

final(self)

Genera l'HMAC de les dades.

data (str)

Capítol 3

RSA

Rivest Shamir Adleman (RSA) [8] és un sistema criptogràfic de clau pública, creat per Ron Rivest, Adi Shamir y Leonard Adleman, utilitzat tant per xifrar dades com per signar-les. RSA basa la seva seguretat en el problema de la factorització entera, per tant, serà segur fins que es trobi un algorisme de factorització ràpid.
RSA consta de dos claus, una clau pública i una clau privada. La clau pública serà compartida amb tothom, mentre que la clau privada solament ha de ser visible pel propietari. Per tant, quan vulguem compartir informació mitjançant RSA haurem de xifrar la informació amb la clau pública del receptor. Gràcies a això, assegurem la confidencialitat de les dades ja que solament poden ser desxifrades pel receptor que tindrà la clau privada.
RSA també es pot utilitzar per signar les dades. En aquest cas l'emissor ha de xifrar les dades amb la seva clau privada i el receptor ha de desxifrar-la amb la clau pública de l'emissor. Així assegurem l'autentificació, la integritat i el no repudi de les dades, però en cap moment assegurem la confidencialitat ja que tothom té accés a la clau pública. Per solucionar això el que s'ha de fer és generar un digest de les dades, signar aquest digest amb la clau privada de l'emissor i finalment, xifrar les dades amb la clau pública del receptor. Tal que el missatge a enviar sigui el següent:

$$(\mathrm{Ku}(\mathrm{Missatge}), \mathrm{Kp}(\mathrm{H}(\mathrm{Missatge})))$$

3.1 Generació de claus RSA

RSA consta dels següents paràmetres:

- L: Longitud de la clau en bits. Es recomana utilitzar una mida superior a 1024 bits.

- p, q: primers aleatoris d'L bits.

- n: mòdul. n = p * q

- totient: φ (n) = ((p – 1) * (q – 1))

- e: exponent públic coprimer amb φ (n). Sol ser 6557.

- d: exponent privat. Tal que d * e = 1 (mod(φ (n))

A partir d'aquests paràmetres obtenim un clau pública (n, e) i una clau privada (d).
La llibreria M2Crypto ens proporciona el mòdul RSA i les següents funcions per generar i carregar claus:

gen_key(bits, e, callback)

Retorna un objecte RSA amb el parell de claus generades.

bits (int)

Longitud de la clau en bits.

e (int)

Exponent públic RSA.

callback

Objecte de tipus callable invocat durant la generació de claus. Per defecte mostra feedback per pantalla.

load_key(file, callback)

Retorna un objecte RSA amb el parell de claus RSA obtingudes del fitxer.

file (str)

Fitxer que conté el parell de claus.

callback

Objecte de tipus callable invocat mentre es carreguen les claus. Per defecte mostra feedback per pantalla.

load_key_string(string, callback)

Retorna un objecte RSA amb el parell de claus RSA obtingudes d'una string.

string (str)

String que conté el parell de claus.

callback

Objecte de tipus callable invocat mentre es carreguen les claus. Per defecte mostra feedback per pantalla.

load_pub_key(file)

Retorna un objecte RSA_pub amb la clau pública obtinguda del fitxer.

file (str)

Fitxer que conté la clau pública.

Aquestes funcions retornen un objecte RSA o RSA_pub del mòdul RSA. Aquestes classes ens proporcionen els següents mètodes per veure i emmagatzemar les claus:

pub(self)

Retorna una tupla amb la clau pública (n, e).

as_pem(self, cipher, callback)

Retorna el parell de claus RSA en format PEM.

cipher (str)

Tipus de xifratge amb el qual xifrarem les dades.

callback

Objecte de tipus callable invocat per obtenir la clau de xifratge. Per defecte es demana per terminal.

save_pem(self, file, cipher, callback)

Emmagatzema el parell de claus RSA en un fitxer en format PEM.

file (str)

Fitxer on emmagatzemarem el parell de claus.

cipher (str)

Tipus de xifratge amb el qual xifrarem les dades.

callback

Objecte de tipus callable invocat per obtenir la clau de xifratge. Per defecte es demana per terminal.

save_pub_key(self, file)

Emmagatzema la clau pública en un fitxer en format PEM.

file (str)

Fitxer on emmagatzemarem la clau pública.

El mòdul EVP ens proporciona la classe PKey on podem emmagatzemar i obtenir el parell de claus RSA amb els següents mètodes:

assign_rsa(self, rsa, capture=1)

Estableix el parell de claus RSA.

rsa

Objecte RSA amb el parell de claus a establir.

capture (int)

Si val 1 obté ambdós claus RSA.

get_rsa(self)

Retorna el parell de claus RSA prèviament establert.

```
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: AES-128-CBC,B890DC5F3F64F0E2F30E9F637FBD3DE5

Fa+EKsA+qS+COzpiyKEZRwpXJMGzZgItgR0AceVffzQDTklHr+3Y/xfHqFT6DaKq
vtIZ0+0+6DN+vGEhYviGRHJDg54d70OywOMTn+VDFcolTj5Ya+GBQbmWqqLHW4qT
ErEkQYtuHpauA1wALW7p+9roJ0Tq0//fgI1zo08ghLoFlcYCZFYnMKdaPDFXbm5h
iKxjiZ7VtSo+CToYPqGo38DOKvPCLJjwaBXhVarKc1Ldphi7cfCXo6bkCYmYvQxd
YyyfXRoPOskVOWiPWbobN/a0LRU+FKfGTk1o2W9HeHnKYdTZNF+yO1CGw7cmt95V
3VuKmqU7uIsl828O+1Q8BXNGV1YpgRxVzomGmbHkY9+43Y1tEqp885Rj4QukBhSR
6WkwbfurJEM5VRmIXneS7gUKkDAmYV09wRNmqPj7AIZdbvtXCRR+L/bAqmXkfeqy
VJOf3J9PgjT2wUtpA0VdeeuEN/NJTDAKXwpPyRfI+glaJz9qBmcfUtJqGAJfXSig
diW2ZxGObIPnSE5ZjBGc20v0GJ7e61WAxzq+N+nTvoDwnal0oS32wsAFVg2HNrmy
36FQxfMCGEDR18TBqtPQALEFUHizla1JZ8xHZfKu7zERtpuaIGFk0IpIf1Tyk+oN
yNPM1Yp9LSMgwR5Se5h/SqRo9hcJE5Cw1GgKYgGwOA7lN1r7Yo2Z5BVfo1BH3w1k
cO17hTMv9tgSgFybmAOZXAZuzW0gJsvYQ5Idu/AfO2UJhv7yc/Xyw7Lrd3jmcUeG
TL8NjWtNROHQ9nQp+F6CFiBBF5e8sMKJtPHtPTHFYSg=
-----END RSA PRIVATE KEY-----
```

Figura 3.1: Claus RSA en format PEM

3.2 Xifrat amb RSA

Com hem mencionat prèviament, amb RSA xifrem amb la clau pública del receptor i desxifrem les dades amb la clau privada de l'emissor. Hem de tenir en compte que solament podem xifrar blocs d'informació inferior a la mida de la clau.
La classe RSA del mòdul RSA de la llibreria M2Crypto ens proporciona els següents mètodes per xifrar dades:

public_encrypt(self, data, padding)

Retorna les dades xifrades amb la clau pública.

data (str)

Dades a xifrar.

padding (int)

Tipus de padding a utilitzar.

private_decrypt(self, data, padding)

Retorna les dades desxifrades amb la clau privada.

data (str)

Dades a desxifrar.

padding (int)

Tipus de padding a utilitzar.

Els tipus de padding que ens proporciona el mòdul RSA són:

1. RSA.pkcs1_padding
2. RSA.sslv23_padding
3. RSA.no_padding
4. RSA.pkcs1_oaep_padding

3.3 Signatura amb RSA

Com hem mencionat prèviament, amb RSA signem amb la clau privada del receptor i verifiquem la signatura amb la clau pública de l'emissor. Si volem mantenir la confidencialitat de les dades hem de signar un digest d'aquestes.
La classe RSA del mòdul RSA de la llibreria M2Crypto ens proporciona els següents mètodes per signar i verificar:

sign(self, digest, algo='sha1')

Signa un digest amb la clau privada i retorna aquesta signatura.

digest (str)

Digest a signar.

algo (str)

Algorisme que s'ha utilitzat a l'hora de crear el digest. Per defecte és sha1.

verify(self, data, signature, algo='sha1')

Verifica la signatura amb la clau pública i retorna True o False.

data (str)

Dades que han estat signades.

signature (str)

Signatura de les dades.

algo (str)

Algorisme que s'ha utilitzat a l'hora de crear el digest. Per defecte és sha1.

També podem utilitzar els següents mètodes si volem signar les dades directament:

private_encrypt(self, data, padding)

Retorna les dades xifrades amb la clau privada.

data (str)

Dades a xifrar.

padding (int)

Tipus de padding a utilitzar.

public_decrypt(self, data, padding)

Retorna les dades desxifrades amb la clau pública.

data (str)

Dades a xifrar.

padding (int)

Tipus de padding a utilitzar.

3.4 Exemple RSA

Donat un emissor A i un receptor B. A vol enviar un missatge xifrat i signat a B.

```
from M2Crypto import RSA, EVP

f = open("encrypted.bin", "w")

a_rsa = RSA.load_key("A_RSAKey.pem")
b_rsa = RSA.load_pub_key("B_RSAPubKey.pem")

message = "Missatge rebut"

encrypted = b_rsa.public_encrypt(data=message,\
 padding=RSA.pkcs1_padding)

messagedigest = EVP.MessageDigest('md5')

messagedigest.update("message")
digest = messagedigest.final()

signature = a_rsa.sign(digest, algo='md5')

f.write(encrypted + signature)
f.close()
```

Listing 3.1: Desxifrar i verificar en RSA

Capítol 4

DSA

Digital Signature Algorithm (DSA) [12] és un estàndard del Govern Federal dels Estats Units per a firmes digitals que es basa en el petit Teorema de Fermat. També consta de dos claus, una pública que serà visible per tothom i una privada que solament serà visible pel propietari. A diferència de RSA, en aquest cas solament podem signar i verificar missatges. Per tant, signarem els missatges amb la clau privada i es verificaran amb la clau pública, en ambdós casos la clau serà la de l'emissor. Així garantim l'autentificació, la integritat i el no repudi de les dades

4.1 Generació de claus DSA

DSA consta dels següents paràmetres:

- L: Nombre de bits del nombre primer (p)
- p: nombre primer aleatori d'L bits (L >= 512) en increments de 64 bits
- q: nombre primer aleatori de 160 bits divisor de (p-1) (Mida del SHA-1)
- g: $g = h^{(p-1)/q} \bmod p$ sent h sencer tal que $1 < h < (p-1)$. Sol escollir-se h=2.

A partir d'aquests paràmetres obtenim una clau pública (p, q, g, $g^x \bmod p$) i una clau privada (x aleatori t.q $1 < x < q$).

La llibreria M2Crypto ens proporciona el mòdul DSA i les següents funcions per generar i carregar claus:

gen_params(bits, callback)

Retorna un objecte DSA amb els paràmetres generats.

bits (int)

Llargada en bits del nombre primer a generar. Si és menor de 512, s'utilitza 512.

callback

Objecte de tipus callable invocat durant la generació dels paràmetres. Per defecte mostra feedback per pantalla.

set_params(p, q, g)

Retorna un objecte DSA amb els paràmetres indicats.

p (str)

Paràmetre p.

q (str)

Paràmetre q.

g (str)

Paràmetre g.

load_params(file, callback)

Retorna un objecte DSA obtenint els paràmetres a partir d'un fitxer en format PEM.

file (str)

Fitxer en format PEM que conté els paràmetres DSA.

callback

Objecte de tipus callable invocat per obtenir la clau de xifratge. Per defecte es demana per terminal.

load_key(file, callback)

Retorna un objecte DSA obtenint les claus a partir d'un fitxer en format PEM.

file (str)

Fitxer en format PEM que conté els paràmetres DSA.

callback

Objecte de tipus callable invocat per obtenir la clau de xifratge. Per defecte es demana per terminal.

load_pub_key(file, callback)

Retorna un objecte DSA_pub amb la clau pública obtinguda a partir d'un fitxer.

file (str)

Fitxer en format PEM que conté la clau pública DSA.

callback

Objecte de tipus callable invocat per obtenir la clau de xifratge. Per defecte es demana per terminal.

Aquestes funcions retornen un objecte DSA o DSA_pub del mòdul DSA. Aquestes classes ens proporcionen els següents mètodes per veure i emmagatzemar les claus:

set_params(self, p, q, g)

Retorna un objecte DSA amb els paràmetres indicats.

p (str)

Paràmetre p.

q (str)

Paràmetre q.

g (str)

Paràmetre g.

gen_key(self)

Genera un nou parell de claus.

save_key(self, filename, cipher='aes_128_cbc', callback)

Desa el parell de claus DSA en format PEM en un fitxer.

filename (str)

Fitxer on guardar el parell de claus DSA.

cipher(str)

Tipus de xifratge amb el qual xifrarem el fitxer. Per defecte utilitza aes_128_cbc.

callback

Objecte de tipus callable invocat per obtenir la clau de xifratge. Per defecte es demana per terminal.

save_pub_key(self, filename)

Desa la clau pública DSA en format PEM en un fitxer.

filename (str)

Fitxer on desar la clau pública DSA.

```
-----BEGIN DSA PARAMETERS-----
MIGdAkEAs5tv1TBEyxBKzxfpEq59aT2lS+/arfJTD2/V0ucyPJnw+PrMnQsM4ZFR
NX1PQHnXcnvrW6+OFhrQWQTelDDwbQIVAPBuUkfd+r8eX7EzHisn8ejYivg7AkEA
qCTFTb7KJB3o6HbhV6xYQhMTHkQO+CshNdZxVzTk6mz6SfFEK0uRU56IDeUKzpDT
5Ff9L05u2UzhnlTdFMB/Zw==
-----END DSA PARAMETERS-----
```

Figura 4.1: Paràmetres DSA en format PEM

4.2 Signatura amb DSA

A l'hora de signar amb DSA es generen dos paràmetres, r i s, que ens serviran posteriorment per verificar la signatura. La forma de calcular aquests paràmetres és la següent:

- Donat un missatge m i una funció de Hash H
- S'escull k aleatori entre 1 i q
- Calculem $r = (g^k \mod p) \mod q$
- Calculem $s = k^{-1}(H(m) + x \cdot r) \mod q$
- Obtenim la firma (r,s)

La classe DSA ens proporciona el següent mètode per signar:

sign(self, digest)

Retorna una tupla (r, s) amb els paràmetres de la signatura.

digest (str)

Digest del missatge a signar.

Un cop rebem el missatge amb la pertinent signatura la forma de verificar-la és la següent:

- Calculem:
  - $w = s^{-1} \mod q$
  - $a = (H(m) \cdot w) \mod q$
  - $b = (r \cdot w) \mod q$
  - $v = ((g^a \cdot (g^x)^b) \mod p) \mod q$

- Si v = r la firma és correcta. Per què?
  - Pel petit Teorema de Fermat. $g^q \equiv 1 \mod p$. g té ordre q.
  - Sabem que $k = (s^{-1} \cdot H(m) + s^{-1} \cdot x \cdot r) \mod q$

La classe DSA ens proporciona el següent mètode per verificar la signatura:

verify(self, digest, r, s)

Verifica la signatura DSA. Retorna 1 si és correcta i 0 en cas contrari.

digest (str)

Digest del missatge rebut.

r (str)

Paràmetre r de la signatura.

s (str)

Paràmetre s de la signatura.

```
from M2Crypto import EVP, DSA

if __name__ == '__main__':
    message = 'Signing and verifing message with DSA'
    md = EVP.MessageDigest('sha1')
    md.update(message)
    digest = md.final()

    dsa = DSA.gen_params(1024)
    dsa.gen_key()
    r, s = dsa.sign(digest)

    verified = dsa.verify(digest, r, s)
    if verified:
        print "Message verified"
    else:
        print "Message not verified"
```

Listing 4.1: Signar i xifrar amb DSA

4.3 Comparació amb RSA

La diferència principal entre DSA i RSA, com ja s'ha mencionat, és que amb RSA es pot xifrar i signar mentre que amb DSA solament podem signar.
En termes de resistència ambdós són equivalents.

			sign	verify	sign/s	verify/s
rsa	512	bits	0.000107s	0.000008s	9359.5	128352.9
rsa	1024	bits	0.000433s	0.000022s	2310.5	46004.2
rsa	2048	bits	0.002426s	0.000074s	412.2	13436.3
rsa	4096	bits	0.017334s	0.000275s	57.7	3634.2

			sign	verify	sign/s	verify/s
dsa	512	bits	0.000091s	0.000095s	11003.1	10512.9
dsa	1024	bits	0.000223s	0.000262s	4486.0	3809.9
dsa	2048	bits	0.000733s	0.000878s	1363.8	1138.4

Taula 4.1: Comparació entre DSA i RSA

L'avantatge principal de DSA és que no està sotmès a restriccions d'exportació al no servir per xifrar. Mentre que l'avantatge principal de RSA és la seva rapidesa a l'hora de verificar en comparació amb DSA.

Capítol 5

Distribució de claus

Com hem vist en els capítols anteriors, tant en el cas de RSA com DSA per tal de poder xifrar o verificar necessitem la clau pública del receptor o emissor. En el cas del xifratge simètric hem de distribuir la clau única entre els dos usuaris Per tant necessitarem una gestió de claus per tal de poder distribuir les claus i que els destinataris puguin obtenir la clau necessària, tot això garantint la seguretat d'aquestes.
En aquest cas diferenciarem entre dos tipus, distribució de claus públiques i distribució de claus secretes.

5.1 Distribució de claus públiques

5.1.1 Anunciament públic

Mitjançant l'anunciament públic, l'usuari distribueix la clau pública als destinataris o l'emet a tota la comunitat. Això es pot dur a terme, per exemple, mitjançant emails o llistes d'emails.
La major feblesa d'aquest mètode és la falsificació ja que qualsevol usuari es pot fer passar per un altre i suplantar la seva identitat a l'hora d'enviar les claus, per tant, la informació no és segura durant la suplantació d'identitat.

5.1.2 Directori disponible públicament

Mitjançant un directori disponible públicament podem obtenir major seguretat registrant les claus. Aquest directori ha de ser de confiança i complir les següents propietats:

- El directori conté entrades amb el nom del propietari de la clau i la clau pública.

- Els participants s'han de registrar de forma segura al directori.
- Els participants poden reemplaçar la clau en qualsevol moment.
- El directori és publicat periòdicament.
- El directori ha de ser accessible de manera electrònica.

Tot i ser més segur encara continua sent vulnerable a falsificacions i manipulacions.

5.1.3 Autoritat de clau pública

L'autoritat de clau pública, com es veu a la Fig. 5.1, millora la seguretat prement el control sobre la distribució de claus des del directori. L'autoritat de clau pública té les propietats d'un directori i requereix que els usuaris coneguin la seva clau pública, d'aquesta manera els usuaris interactuen en temps real amb el directori per tal d'obtenir la clau pública desitjada de manera segura.
Tot això és completament segur partint de la premissa de que podem confiar en l'autoritat de clau pública i no ha estat suplantada, en cas contrari, aquest mètode no és segur ja que no podem garantir la veracitat de les claus que rebem.

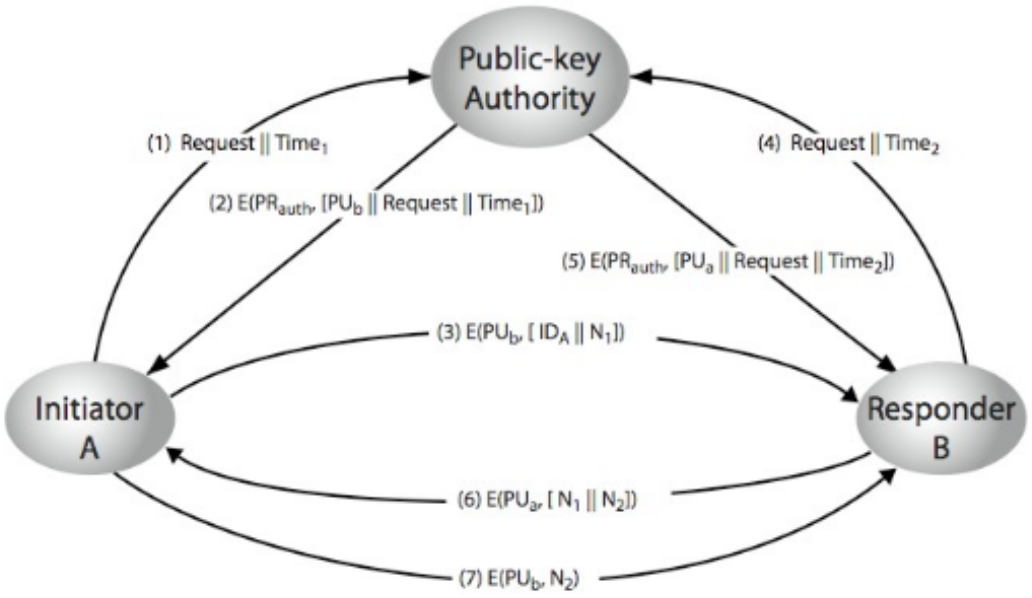

Figura 5.1: Distribució de claus públiques mitjançant una autoritat de clau pública. Figura extreta de Cryptography and Network Security Principles and Practices [17]

5.1.4 Certificats de clau pública

Mitjançant certificats de clau pública, com es veu a la Fig. 5.2, permetem l'intercanvi de claus sense necessitat d'interactuar en temps real amb l'autoritat de clau pública. Els certificats enllacen l'identitat amb la clau pública juntament amb més informació com el període de validesa, els drets d'ús, etc. Amb tots els continguts signats amb una clau pública o Autoritat Certificadora (CA) de confiança. D'aquesta manera la CA genera un certificat amb la clau pública que pot ser verificat per qualsevol que conegui la clau pública de la CA.
Com hem esmentat, aquest mètode solament és segur si podem assegurar la confiança de la CA.

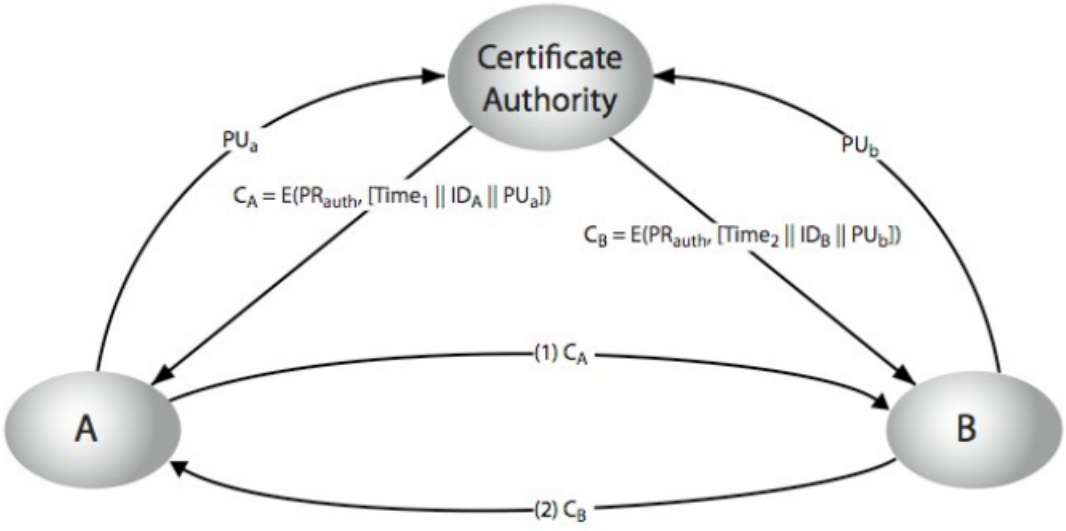

Figura 5.2: Distribució de claus públiques mitjançant certificats de clau pública. Figura extreta de Cryptography and Network Security Principles and Practices [17]

5.2 Distribució de claus secretes

5.2.1 Distribució de clau simètrica

Mitjançant la distribució de clau simètrica s'estableix una clau mestra per entre els dos participants. Diferenciem entre dos tipus:

- Centralitzat: es generen n claus mestres, on n és el nombre d'usuaris, com es veu a la Fig. 5.3.

- Distribuït: es generen $n(n-1)/2$ claus mestres, on n és el nombre d'usuaris, com es veu a la Fig. 5.4.

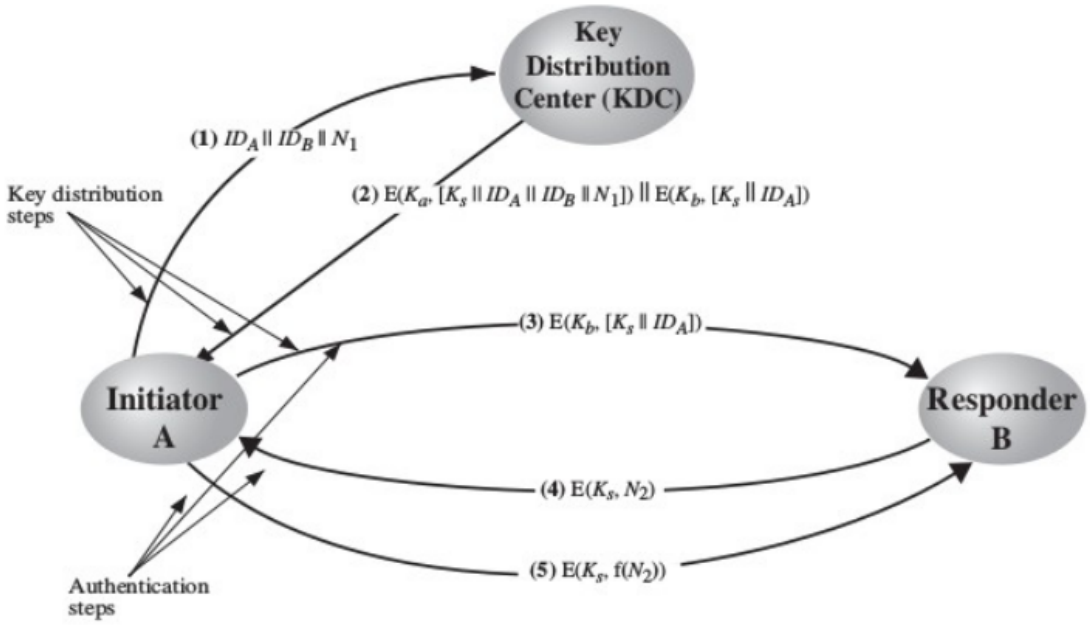

Figura 5.3: Distribució de clau secreta mitjançant clau simètrica centralitzada. Figura extreta de Cryptography and Network Security Principles and Practices [17]

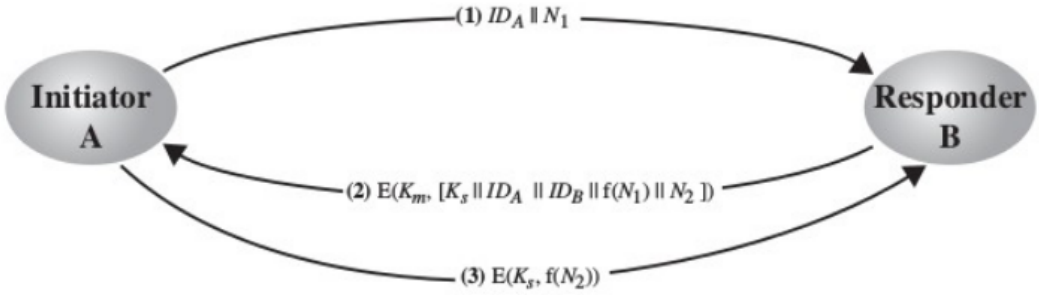

Figura 5.4: Distribució de clau secreta mitjançant clau simètrica distribuïda. Figura extreta de Cryptography and Network Security Principles and Practices [17]

5.2.2 Distribució de clau pública

Per tal de poder distribuir claus secretes mitjançant la distribució de clau pública, com es veu a la Fig. 5.5, primer els participants han d'intercanviar les claus

públiques de forma segura mitjançant els mètodes explicats prèviament. D'aquesta manera els participants podran verificar la identitat del receptor i xifraran la clau secreta amb la clau pública d'aquest. L'inconvenient és que tots aquests passos són bastant lents.

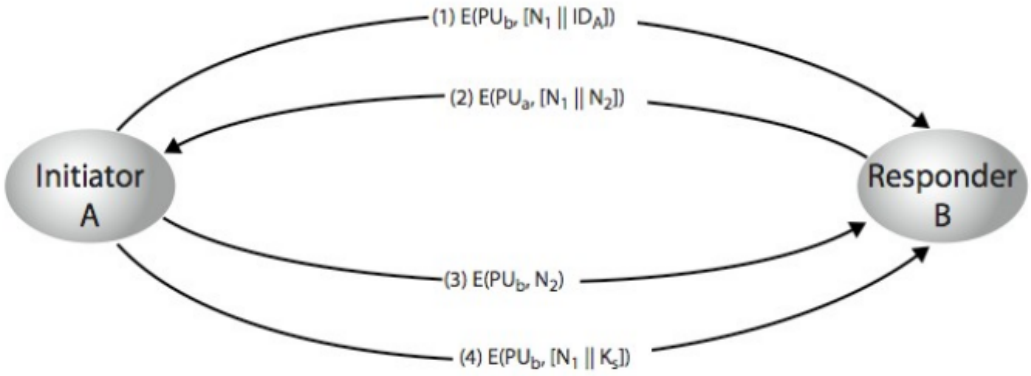

Figura 5.5: Distribució de clau secreta mitjançant clau pública. Figura extreta de Cryptography and Network Security Principles and Practices [17]

5.2.3 Distribució de clau secreta simple

La distribució de clau secreta simple va ser proposada per Merkle l'any 1979 i consisteix en el següent:

- A genera un nou parell de claus públiques temporal.
- A envia a B la clau pública i la seva identitat.
- B genera una clau de sessió K i l'envia a A encriptada amb la clau pública obtinguda.
- A desxifra la clau de sessió K i ambdós l'utilitzen per xifrar i desxifrar els següents missatges.

L'inconvenient d'aquest mètode és que es pot suplantar l'identitat tant d'A com de B.

5.2.4 Distribució de clau híbrida

La distribució de clau híbrida conserva l'ús de clau privada KDC, comparteix una clau mestra amb cada usuari i distribueix la clau de sessió utilitzant aquesta clau mestra. S'utilitza la clau pública per tal de distribuir les claus mestres, especialment útil amb un ampli número d'usuaris.
D'aquesta manera augmentem el rendiment i és compatible amb versions anteriors.

5.2.5 Diffie-Hellman

Diffie-Hellman és el primer esquema de clau pública proposat per Whitfield Diffie y Martin Hellman l'any 1976. Tot i que ara sabem que el concepte va ser proposat secretament per Malcolm J. Williamson l'any 1970 quan treballava per als serveis d'intel·ligència i seguretat britànics.
Aquest és un mètode pràctic per a l'intercanvi públic d'una clau secreta utilitzat freqüentment en productes comercials. Estableix una clau compartida, a partir de la informació de les claus públiques i privades, que solament coneixeran els dos participants, per tant, no es pot utilitzar solament per enviar un missatge arbitrari.
Aquest esquema es basa en l'exponenciació en un camp finit (Galois) i la seva seguretat recau en la dificultat de computar logaritmes discrets, similar a la factorització.

Generació de claus de Diffie-Hellman

A l'hora de generar les claus hem de seguir els següents passos:

- Tots els usuaris acorden els paràmetres globals.
  - q: Gran enter primer o polinòmic.
  - g: Arrel primitiva mòdul q.
- Cada usuari genera les seves claus.
  - x: Escolleix una clau privada tal que x $<$ q.
  - g^a : Calcula la clau pública $g^a = g^a \mod q$
- Cada usuari fa pública la clau g^x

La llibreria M2Crypto ens proporciona el mòdul DH i les següents funcions per tal de generar i carregar els paràmetres i les claus:

gen_params(plen, g, callback)

Genera els paràmetres globals de Diffie-Hellman.

plen (int)

Llargada en bits del paràmetre p.

g (str)

Paràmetre g.

callback

Objecte de tipus callable.

load_params(file)

Càrrega els paràmetres de Diffie-Hellman des d'un fitxer en format PEM.

file (str)

Fitxer en format PEM que conté els paràmetres.

set_params(p, g)

Estableix els paràmetres de Diffie-Hellman.

p (str)

Paràmetre p.

g (str)

Paràmetre g.

Intercanvi de claus Diffie-Hellman

Per tal de generar la clau compartida entre els dos usuaris s'utilitza la següent operació:

$$K_s = (g^{x_a})^{x_b} \mod q = g^{x_a \cdot x_b} \mod q$$

És a dir s'eleva la clau pública del receptor per la nostra clau privada. Així obtenim la clau de sessió per poder xifrar i desxifrar les dades entre ambdós usuaris utilitzant xifrat simètric. Mentre cap dels usuaris modifiqui la clau pública, la clau de sessió continuarà sent la mateixa.
Diffie-Hellman és vulnerable a atacs man-in-the-middle, ja que un atacant que conegui els paràmetres es pot situar entre ambdós participants i falsificar l'identitat d'ambdós. D'aquesta manera l'atacant enviaria la seva clau pública a A fent-se passar per B i viceversa. Això es pot solucionar enviant les claus públiques signades per autoritats fiables.

A partir del següent exemple podem veure tot el procés per generar la clau compartida Diffie-Hellman:

- Els usuaris Alice i Bob volen intercanviar claus.
- Acorden com a primer q = 353 i g = 3.
- Escolleixen les claus secretes.
  - Alice escolleix $x_a = 97$
  - Bob escolleix $x_b = 233$
- Calculen les respectives claus públiques.
  - $g^a = 3^{97} \mod 353 = 40$ (Alice)
  - $g^b = 3^{233} \mod 353 = 248$ (Bob)
- Finalment, generen la clau compartida.
  - $K_s = (g^b)^{x_a} \mod 353 = 248^{97} \mod 353 = 160$ (Alice)
  - $K_s = (g^a)^{x_b} \mod 353 = 40^{233} \mod 353 = 160$ (Bob)

La classe DH del mòdul DH ens proporciona els següents mètodes per tal de generar les claus, pública i privada, i la clau compartida.

gen_key(self)

Genera la clau pública i privada.

compute_key(self, pubkey)

Genera la clau compartida Diffie-Hellman.

pubkey (str)

Clau pública del receptor a partir de la qual es genera la clau compartida.

Exemple Diffie-Hellman

Donats els paràmetres acordats en un fitxer PEM i la clau pública de B en hexadecimal, generar la clau compartida i xifrar les dades amb aquesta clau per tal de que solament B la pugui desxifrar amb la nostra clau pública.

```
from M2Crypto import DH
from M2Crypto.EVP import Cipher

def read_pub_key(filename):
    f = open(filename)
    key = f.read().split(' ')[2].strip().decode('hex')
    f.close()
    return key

def cipher(message, algo, passwd):
    c = Cipher(alg=algo, key=passwd, iv = "", \
    op=1, padding=1)
    m = c.update(message)
    m += c.final()
    return m


if __name__ == "__main__":
    message = "Encrypting with DH shared key"
    b_pubkey = read_pub_key("pubkey.asc")
    a = DH.load_params("dhpar.pem")
    a.gen_key()
    K_s = a.compute_key(b_pubkey)
    cipher_message = cipher(message,'des_ede_cbc', K_s)
    print cipher_message
```

Listing 5.1: Diffie-Hellman

Capítol 6

Certificats Digitals

6.1 Servei d'autenticació X.509

X.509 [5] és un estàndard per infraestructures de clau pública que forma part del estàndards de servei de directori X.500 [19]. LA seva sintaxi es defineix utilitzant el llenguatge Abstract Syntax Notation One (ASN.1) [10] i els formats de codificació més comuns són PEM i DER.
Aquest estàndard defineix un servei d'autentificació on emmagatzema en un directori els certificats de clau pública amb la clau pública signada per la CA. També defineix protocols d'autenticació.
Utilitza criptografia de clau pública i signatura digital, els algorismes no estan estandarditzats però es recomana utilitzar RSA.

6.2 Certificats X.509

En X.509 un CA emet un certificat associant una clau pública a un nom particular. L'estructura dels certificats, com es veu a la Fig. 6.1, és la següent:

- Versió (1, 2 o 3).
- Nombre de sèrie únic dins de la CA identificant el certificat.
- Identificador de l'algorisme de signatura.
- Nom de l'emissor (CA).
- Període de validesa del certificat.
- Nom del propietari del certificat.

- Informació de la clau pública del propietari del certificat (algorisme, paràmetres, clau)
- Identificador únic de l'emissor (v2+).
- Identificador únic del propietari (v2+).
- Camps d'extensió (v3).
- Signatura del hash de tots els camps del certificat.

Qualsevol usuari que tingui accés a la CA pot obtenir un certificat d'aquesta. Aquest certificat solament pot ser creat i modificat per la CA, per tant, no es pot falsificar i pot estar ubicat en un directori públic.
Si dos usuaris comparteixen la mateixa CA, s'assumeix que ambdós coneixen la seva clau pública, d'altra banda, les CA's han de formar una jerarquia on s'utilitzen certificats que uneixen els membres de la jerarquia per poder validar altres CA's. Aquesta jerarquia és un arbre on cada CA té un certificat pare i certificats per a clients (fills), on cada client confia en els certificats pares. L'anotació CA<<A>>indica que el certificat A ha estat signat per CA. Com es veu a la Fig. 6.2, si A volgués verificar el certificat de B necessita R<<l1>>mentre que B necessita R<<l1>>i l1<<l2>>.
Cada certificat té un període de validesa, per tant, un cop ha expirat hem de revocar aquest certificat. Per això cada CA ha de mantenir una llista de certificats revocats anomenada Certificate Revocation List (CRL), com es veu a la Fig. 6.3. Els certificats també poden ser revocats abans d'expirar, per exemple, si es veu compromesa la clau privada. Per tant abans de confiar en un certificat s'ha de comprovar que no estigui revocat a la CRL de la CA.

La llibreria M2Crypto ens proporciona el mòdul X509 i les següents funcions per carregar certificats i peticions de certificat:

load_cert(file, format=1)

Carrega el certificat del fitxer en un objecte X509.

file (str)

Fitxer que conté el certificat.

format (int)

Format del fitxer, 0 si és DER i 1 si és PEM. Per defecte PEM.

load_cert_string(string, format=1)

Carrega el certificat de la string en un objecte X509.

string (str)

String que conté el certificat.

format (int)

Format del certificat, 0 si és DER i 1 si és PEM. Per defecte PEM.

load_request(file, format=1)

Carrega la petició de certificat dins d'un objecte Request.

file (str)

Fitxer que conté la petició de certificat.

format (int)

Format de la petició de certificat, 0 si és DER i 1 si és PEM. Per defecte PEM.

load_request(string, format=1)

Carrega la petició de certificat dins d'un objecte Request.

string (str)

String que conté la petició de certificat.

format (int)

Format de la petició de certificat, 0 si és DER i 1 si és PEM. Per defecte PEM.

També ens proporciona la següent funció per crear extensions:

new_extension(name, value)

Genera un objecte X509_Extension amb l'extensió indicada.

name (str)

Nom de l'extensió.

value (str)

Valor de l'extensió.

6.3 Procediments d'autenticació

X.509 inclou tres procediments d'autenticació que utilitzen signatura de clau pública.

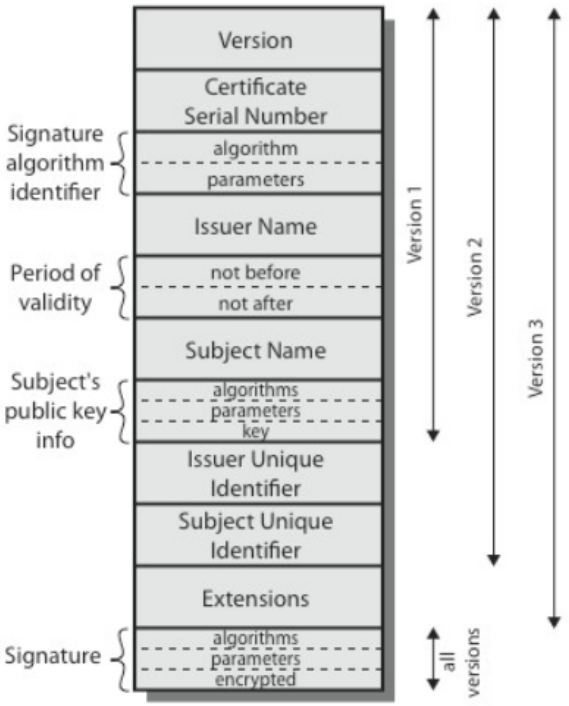

Figura 6.1: Estructura certificat X.509. Figura extreta de Cryptography and Network Security Principles and Practices [17]

6.3.1 One-Way Authentication

S'envia 1 missatge, d'A cap a B, per tal d'establir la identitat d'A, que el missatge prové d'A, que aquest missatge va dirigit a B i la integritat i originalitat d'aquest missatge. Aquest missatge ha d'incloure la data i hora actuals, un nombre arbitrari aleatori (nonce) i la identitat de B, tot això signat per A. També pot incloure informació addicional com la clau de sessió.

6.3.2 Two-Way Authentication

S'envien 2 missatges, d'A cap a B i viceversa, per tal d'establir la identitat de B, que la resposta procedeix de B i va destinada cap a A i la integritat i originalitat de la resposta. La resposta ha d'incloure el nombre arbitrari aleatori (nonce) enviat per A, la data i hora actuals i un nonce generat per B, també pot incloure informació addicional.

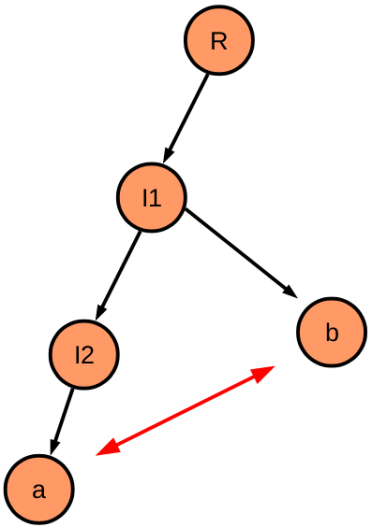

Figura 6.2: Jerarquia de certificats. Figura extreta de Transparències Seguretat d'Aplicacions i Comunicacions [2]

6.3.3 Three-Way Authentication

S'envien 3 missatges, d'A cap a B, de B cap a A i d'A cap a B, per tal d'establir l'autenticació sense necessitat de rellotges sincronitzats, per tant ja no es comprova la data i hora dels missatges. La resposta d'A cap a B conté el nombre arbitrari aleatori (nonce) generat per B.

6.4 Creació de certificats X.509

Per tal de poder generar un certificat, primerament hem de generar una clau privada i una clau pública, s'acostuma a utilitzar RSA, després generar una petició de certificat amb les nostres dades i la clau pública generada i, finalment, s'envia aquesta petició de certificat a una CA per tal de que ens generi el certificat i el signi. El mòdul X509 ens proporciona la classe X509_Name per tal d'emmagatzemar les nostres dades i utilitzar-les posteriorment. Els atributs més habituals d'aquesta classe són:

- C (14): Country
- CN (13): Common Name
- Email (48): Email
- L (15): Locality and address

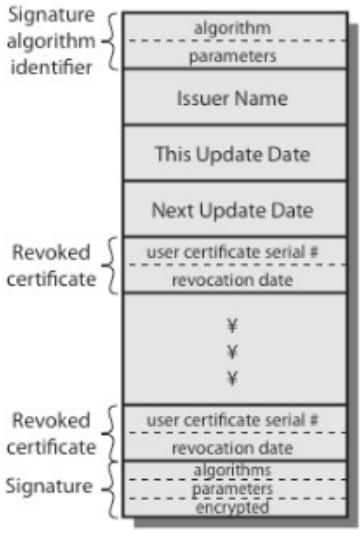

Figura 6.3: Estructura CRL. Figura extreta de Cryptography and Network Security Principles and Practices [17]

- O (17): Organization
- OU (18): Organization Unit
- SN (100): Surname

Per tal de modificar o veure aquests atributs podem accedir directament o utilitzar els següents mètodes:

add_entry_by_txt(self, field, type, entry, len, loc, set)

Afegeix l'entrada indicada.

field (str)

Camp el qual es vol afegir.

type (int)

Tipus de les dades a inserir.

entry (str)

Entrada a afegir.

len (int)

Longitud en bytes de l'entrada a inserir. Si val -1 es calcula automàticament.

loc (int)

Índex on afegir la nova entrada. Si val -1 s'annexa.

set (int)

Determina com s'afegeix l'entrada. Si val 0 es crea una nova, mentre que si val -1 o 1 s'afegeix a la prèvia o següent.

get_entries_by_nid(self, nid)

Retorna l'entrada a partir del seu NID.

nid (int)

NID de l'entrada.

Per tal de crear i emmagatzemar peticions de certificat el mòdul X509 ens proporciona la classe Request amb els següents mètodes:

save_pem(self, filename)

Emmagatzema la petició de certificat en un fitxer en format PEM.

filename (str)

Fitxer on emmagatzemar la petició.

save(self, filename, format=1)

Emmagatzema la petició de certificat en un fitxer en el format indicat.

filename (str)

Fitxer on emmagatzemar la petició.

format (int)

Format del fitxer, 0 si és DER i 1 si és PEM. Per defecte PEM.

set_pubkey(self, pkey)

Estableix la clau pública a utilitzar.

pkey

Objecte EVP.PKey amb la clau pública.

get_pubkey(self)

Retorna un objecte EVP.PKey amb la clau pública.

set_version(self, version)

Estableix la versió de X509 que volem emprar.

version (int)

Versió a utilitzar.

get_version(self)

Retorna la versió de X509 establerta.

set_subject(self, name)

Estableix les dades del propietari de la petició.

name

Objecte de tipus X509_Name amb les dades del propietari de la petició.

get_subject(self, name)

Retorna les dades del propietari de la petició.

sign(self, pkey, md)

Signa el hash de la petició amb la clau privada.

pkey

Objecte de tipus EVP.PKey que conté la clau privada.

md (str)

Funció de hash a utilitzar.

verify(self, pkey)

Verifica la signatura del hash de la petició amb la clau pública.

pkey

Objecte de tipus EVP.PKey que conté la clau pública.

El mòdul X509 ens proporciona la classe X509, per tal de crear i manipular certificats X509, amb els següents mètodes:

save_pem(self, filename)

Guarda el certificat en format PEM en el fitxer indicat.

filename (str)

Fitxer on emmagatzemar el certificat.

save(self, filename, format=1)

Guarda el certificat en el format indicat en un fitxer.

filename (str)

Fitxer on emmagatzemar el certificat.

format (int)

Format del fitxer, 0 si és DER i 1 si és PEM. Per defecte PEM.

set_version(self, version)

Estableix la versió de X509 que volem emprar.

version (int)

Versió a utilitzar.

get_version(self)

Retorna la versió de X509 establerta.

set_serial_number(self, serial)

Estableix el nombre de sèrie del certificat.

serial (int)

Nombre de sèrie.

get_serial_number(self)

Retorna el nombre de sèrie del certificat.

set_issuer(self, name)

Estableix les dades de la CA signant.

name

Objecte de tipus X509_Name amb les dades de la CA signant.

get_issuer(self, name)

Retorna les dades de la CA signant.

set_subject(self, name)

Estableix les dades del propietari del certificat.

name

Objecte de tipus X509_Name amb les dades del propietari del certificat.

get_subject(self, name)

Retorna les dades del propietari del certificat.

set_pubkey(self, pkey)

Estableix la clau pública a utilitzar.

pkey

Objecte EVP.PKey amb la clau pública.

get_pubkey(self)

Retorna un objecte EVP.PKey amb la clau pública.

add_ext(self, ext)

Afegeix una extensió al certificat.

ext

Objecte X509_Extension que conté l'extensió.

get_ext(self, name)

Retorna l'extensió a partir del seu nom.

name

Nom de l'extensió.

sign(self, pkey, md)

Signa el hash del certificat amb la clau privada.

pkey

Objecte de tipus EVP.PKey que conté la clau privada.

md (str)

Funció de hash a utilitzar.

verify(self, pkey)

Verifica la signatura del hash del certificat amb la clau pública.

pkey

Objecte de tipus EVP.PKey que conté la clau pública.

set_not_before(self, asn1_utctime)

Estableix la data a partir de la qual el certificat és vàlid.

asn1_utctime

Objecte ASN1_UTCTIME amb la data.

set_not_after(self, asn1_utctime)

Estableix la data a partir de la qual el certificat deixa de ser vàlid.

asn1_utctime

Objecte ASN1_UTCTIME amb la data.

check_ca(self)

Comprova si el certificat pertany a una CA.

get_fingerprint(self, md='md5')

Retorna l'empremta digital del certificat en format hexadecimal.

md (str)

Funció de hash a utilitzar. Per defecte MD5.

Per tal de poder obtenir la data en format ASN.1 la llibreria M2Crypto ens proporciona la classe ASN1_UTCTIME, del mòdul ASN1, amb els següents mètodes:

set_string(self, string)

Estableix la data a partir d'una string amb la data en format UTC.

string (str)

Data en format UTC.

set_time(self, time)

Estableix la data en segons des de l'1 de gener de 1970 a les 00:00:00 (epoch).

string (str)

Data en segons.

set_datetime(self, date)

Estableix la data.

string (str)

Data a establir.

get_datetime(self)

Retorna la data establerta.

A partir d'aquestes classes i mètodes, en el codi 6.1 generarem la nostra propia jerarquia, com es veu a la Fig. 6.4, que utilitzarem en els següents apartats on veurem SSL i les seves aplicacions.

```
from M2Crypto import X509, EVP, RSA, ASN1, BN
import time

def create_request(bits, cn, mail):
    req = X509.Request()
    pk = EVP.PKey()
    rsa = RSA.gen_key(bits, 65537)
    rsa.save_pem(cn + 'key.pem')
    pk.assign_rsa(rsa)
    req.set_pubkey(pk)
    name = req.get_subject()
    name.C = "ES"
    name.O = "Default Company Ltd"
    name.CN = cn
    name.ST = 'CA'
    name.L = 'Lleida'
    name.Email = mail
    req.sign(pk, 'sha1')
    return req

def mk_add_validity_period(cert, days=365):
    t = long(time.time())
    now = ASN1.ASN1_UTCTIME()
    now.set_time(t)
    expire = ASN1.ASN1_UTCTIME()
    expire.set_time(t + days * 24 * 60 * 60)
    cert.set_not_before(now)
    cert.set_not_after(expire)


def create_signed_cert(req, pk, ca_cert=None,\
 is_ca=False, is_autosigned=False):
    pubkey = req.get_pubkey()
    cert = X509.X509()
    cert.set_version(req.get_version())
    mk_add_validity_period(cert)
    if is_autosigned:
        cert.set_issuer(req.get_subject())
        cert.set_serial_number(1)
    else:
        cert.set_issuer(ca_cert.get_subject())
        cert.set_serial_number(ca_cert.get_serial_number()\
         + BN.rand(128))
    cert.set_subject(req.get_subject())
    cert.set_pubkey(pubkey)
```

```
    if is_ca == True:
        cert.add_ext(
        X509.new_extension('basicConstraints', 'CA:TRUE'))
    else:
        cert.add_ext(
        X509.new_extension('basicConstraints', 'CA:FALSE'))
    cert.add_ext(
    X509.new_extension('subjectKeyIdentifier',\
     cert.get_fingerprint()))
    cert.sign(pk, 'sha1')
    return cert

def generate_certificate(ca, cn, ca_filename, mail, is_ca):
    ca_cert = X509.load_cert(ca_filename)
    req = create_request(1024, cn, mail)
    req.save_pem(cn + 'req.pem')
    rsa = RSA.load_key(ca + 'key.pem')
    pk = EVP.PKey()
    pk.assign_rsa(rsa)
    cert = create_signed_cert(req, pk, ca_cert=ca_cert,\
     is_ca=is_ca)
    cert.save_pem(cn +'cert.pem')

def generate_autosigned_certificate(cn, mail):
    req = create_request(1024, cn, mail)
    req.save_pem(cn + 'req.pem')
    rsa = RSA.load_key(cn + 'key.pem')
    pk = EVP.PKey()
    pk.assign_rsa(rsa)
    rootcert = create_signed_cert(req, pk, is_ca=True,\
     is_autosigned=True)
    rootcert.save_pem(cn + 'cert.pem')

if __name__ == "__main__":
    generate_autosigned_certificate('root',\
     'acm24@alumnes.udl.cat')
    generate_certificate('root', 'serverCA',\
     'rootcert.pem', 'acm24@alumnes.udl.cat', True)
    generate_certificate('serverCA', 'server',\
     'serverCAcert.pem', 'acm24@alumnes.udl.cat', False)
    generate_certificate('root', 'client',\
     'rootcert.pem', 'acm24@alumnes.udl.cat', False)
```

Listing 6.1: Generació de certificats

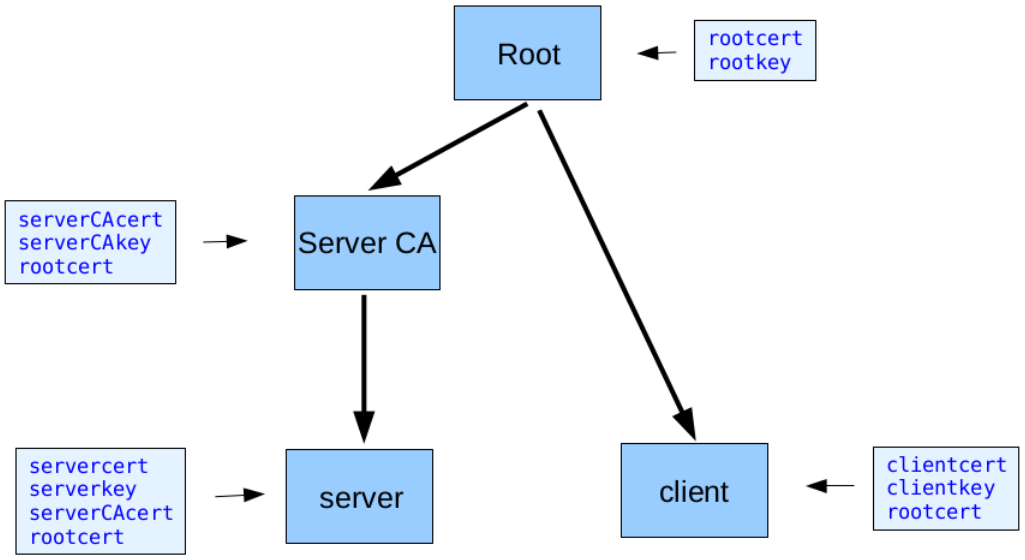

Figura 6.4: Jerarquia de certificats a generar. Figura extreta de Transparències Seguretat d'Aplicacions i Comunicacions [2]

Un cop hem generat els certificats utilitzarem la comanda cat per unir tota la cadena de certificació, per exemple per al certificat server haurem de fer-ho de la següent manera:

cat servercert.pem serverkey.pem serverCAcert.pem rootcert.pem > server.pem

6.5 Emmagatzemament de certificats

Per tal de poder emmagatzemar els certificats el mòdul X509 ens proporciona les classes X509_Store i X509_Stack. La classe X509_Store emmagatzema els certificats sense tenir en compte la posició i ens proporciona els següents mètodes:

load_locations(self, file)

Carrega a l'store la cadena de certificació.

file (str)

Fitxer que conté la cadena de certificació.

add_x509(self, x509)

Carrega a l'store el certificat.

x509

Certificat X509.

Mentre que la classe X509_Stack emmagatzema els certificats en una pila i ens proporciona els següents mètodes.

push(self, x509)

Afegeix el certificat al cim de la pila.

x509

Certificat X509.

pop(self)

Extreu el certificat del cim de la pila.

Capítol 7

SSL

La principal funcionalitat d'OpenSSL és la implementació de protocols SSL i TLS. Originalment desenvolupats per Netscape per a transaccions web segures, el protocol ha anat creixent fins a convertir-se en una solució per a comunicacions segures. La primera versió pública de Netscape d'SSL és el que coneixem com a SSL Versió 2. A partir d'aquesta versió els experts en seguretat han treballat en millorar-la donant com a resultat SSL Versió 3. Paral·lelament, es desenvolupa un estàndard de capa de transport segura basat en SSL donant com a resultat TLS Versió 1. Donat els defectes de SSLv2, les aplicacions modernes no suporten aquest protocol. Per tant, en aquest capítol programarem amb els protocols SSLv3 i TLSv1, tot i que, a l'hora d'escollir el més segur ens hem de decantar per TLSv1. Quan parlem d'aquests protocols ens referirem a tots dos com SSL.
Per a dur a terme aquesta aplicació utilitzarem els següents mòduls de M2Crypto: SSL i m2 i utilitzarem la jerarquia de certificats creada en el capítol 6, comes veu a la Fig. 6.4..

7.1 Aplicació segura

Utilitzarem dos aplicacions simples: un client i un servidor, on el servidor rebrà dades del client i les mostrarà per consola. L'objectiu és mostrar els mecanismes de seguretat i com van augmentant per tal que l'aplicació pugui dur a terme la seva tasca en un ambient hostil.
Per tal de fer aquesta connexió segura haurem de completar cadascun dels passos del Handshake SSL correctament i amb l'informació adequada 7.1.
L'aplicació client, client.py [7.1]. Primerament el client crea un context, seguidament es crea la connexió i es connecta al port 2016 del servidor i finalment envia les dades que ha d'escriure el servidor.

L'aplicació servidor, server.py [7.2]. Primerament el servidor crea un context i es carrega el certificat i la clau privada del certificat del servidor. Seguidament es crea una connexió que es deixa escoltant al port 2016. I finalment s'espera a rebre una connexió i quan es rep es llença un thread on llegirà la informació enviada pel client i la mostrarà per consola.
Aquesta és la versió inicial de l'aplicació la qual anirem millorant en els següents punts per tal d'assegurar la comunicació mitjançant SSL. Amb aquesta versió codifiquem les dades que s'envien però no podem assegurar en cap moment qui hi ha a l'altra banda de la connexió ni controlem quin protocol o quin tipus de xifratge s'utilitza a l'hora d'establir la connexió.

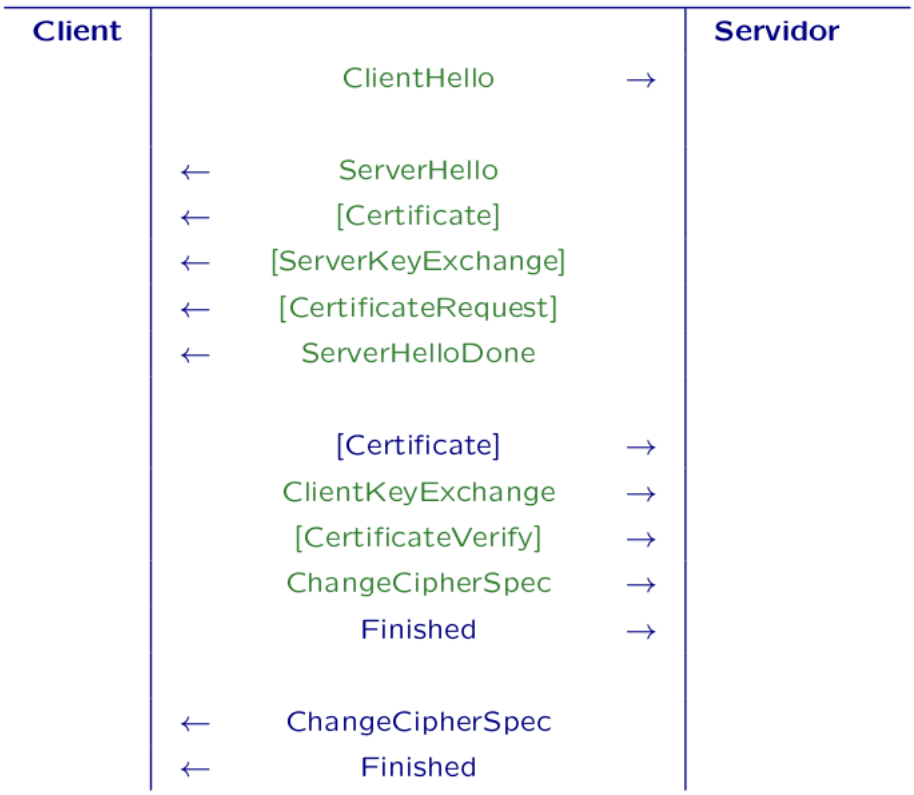

Figura 7.1: Handshake SSL

```
from M2Crypto import SSL

def init_context(protocol):
    ctx = SSL.Context(protocol)
    return ctx
```

```
if __name__ == "__main__":
    protocol = "sslv23"

    ctx = init_context(protocol)

    SSL.Connection.clientPostConnectionCheck = None
    conn = SSL.Connection(ctx)
    conn.connect(("127.0.0.1", 2016))

    message = raw_input("Insert text to send\n")

    conn.write(message)

    conn.close()
```

Listing 7.1: Client SSL 1

```
from M2Crypto import SSL
import sys
from threading import Thread

def init_context(protocol, certfile, certkey):
    ctx = SSL.Context("tlsv1")
    ctx.load_cert(certfile=certfile, keyfile=certkey)
    ctx.set_info_callback()
    return ctx

def get_message(ssl):
    msg = ssl.read()
    print msg
    ssl.close()


if __name__ == "__main__":
    certfile = "servercert.pem"
    certkey = "serverkey.pem"

    addr = ("127.0.0.1", 2016)
    protocol = "sslv23"

    ctx = init_context(protocol, certfile, certkey)

    conn = SSL.Connection(ctx)

    try:
        conn.bind(addr)
```

```
except:
    print "Bind failed"
    sys.exit(-1)

conn.listen(10)

while 1:
    ssl, addr = conn.accept()
    print "Connected with " + addr[0]
    Thread(target=get_message, args=(ssl,)).start()

conn.close
```

Listing 7.2: Servidor SSL 1

7.2 Selecció de protocol i preparació de certificats

Per tal de que la nostra aplicació sigui més segura hem de poder establir quin tipus de protocol es podrà utilitzar per tal d'establir la connexió, ja que com hem mencionat anteriorment hem de prioritzar l'ús de SSLv3 i TLSv1 davant de SSLv2. Un altre apartat per tal de millorar la seguretat és poder garantir al client l'autenticació del servidor i poder garantir l'autenticació del client davant del servidor, per tal d'evitar atacs man-in-the-middle.

7.2.1 Selecció de protocol

A l'hora d'indicar quin tipus de protocol establim ho hem de fer al crear l'objecte Context mitjançant el paràmetre protocol, l'especificació del constructor és la següent:

SSL.Context(self, protocol='sslv23', weak_crypto=None)

protocol (str)

Tipus de protocol que es permetrà per establir connexions.
Els diferents valors de protocol són:

'sslv23': Es permeten tot tipus de connexió tant SSLv2 i SSLv3 com TLSv1. Aquest valor s'estableix per defecte.

'sslv3': Solament es permeten connexions de tipus SSLv3.

'tlsv1': Solament es permeten connexions de tipus TLSv1.

weak_crypto (bool)

Establim si es permeten xifratges dèbils. Per defecte sí es permeten.

7.2.2 Preparació de certificats

El protocol SSL normalment requereix que el servidor presenti un certificat. El certificat conté les credencials amb les quals el client podrà comprovar si el servidor està autenticat i es pot confiar. La verificació del certificat es realitza amb la cadena de signants. Per tant per implementar el servidor correctament hem de proporcionar el certificat i la cadena de certificats.
El protocol SSL també ens permet que el client presenti la seva cadena de certificació per tal de que el servidor l'autentiqui.
Per tal de que tant el client com el servidor puguin verificar els certificats haurem d'establir la Certification Authority (CA) a partir de la qual han estat generats.
El certificat l'establirem dins de l'objecte Context amb la següent funció:

SSL.Context.load_cert(self, certfile, keyfile=None, callback)

Carrega el certificat.

certfile(str)

Fitxer que conté el certificat en format PEM.

keyfile (str)

Fitxer que conté la clau privada en format PEM. Per defecte estem indicant que la clau privada es troba dins de certfile.

callback

Objecte de tipus callable invocat per obtenir la clau de xifratge del fitxer. Per defecte la clau es demana per terminal.

Si volem establir la cadena de certificació utilitzarem la següent funció:

SSL.Context.load_cert_chain(self, certfile, keyfile=None, callback)

Carrega la cadena de certificació.

certfile (str)

Fitxer que conté la cadena de certificació en format PEM.

keyfile (str)

Fitxer que conté la clau privada en format PEM. Per defecte estem indicant que la clau privada es troba dins del certfile.

callback

Objecte de tipus callable invocat per obtenir la clau de xifratge del fitxer. Per defecte la clau es demana per terminal.

Per tal d'establir les CA per validar els certificats utilitzarem la següent funció:

SSL.Context.load_verify_locations(self, cafile=None, capath=None)

Estableix les CA amb les quals verificar els certificats.

cafile (str)

Fitxer que conté un o més certificats de CA concatenats en format PEM.

capath (str)

Directori que conté un o més certificats de CA concatenats en format PEM.

Per tal d'indicar que s'ha de verificar el certificat del client o del servidor hem d'utilitzar la següent funció:

SSL.Context.set_verify(self, mode, depth, callback=None)

Estableix la verificació dels certificats.

mode (int)

Mode de verificació a utilitzar.

Els diferents valors de mode són:

- *SSL.verify_none = 0*: En la part de servidor, no envia cap petició de certificat al client i aquest no envia el seu certificat. En la part de client, qualsevol certificat enviat pel servidor serà verificat, però no fallarà la fase de handshake si no es verifica correctament.
- *SSL.verify_peer = 1*: En la part de servidor, s'envia al client una petició de certificat. El client pot optar per ignorar-la, però si envia el certificat aquest serà verificat i en cas de que no es verifiqui correctament la fase de handshake es donarà per acabada immediatament.
  En la part de client, el certificat del servidor serà verificat i en cas de que no es verifiques correctament la fase de handshake es donarà per finalitzada immediatament.
- *SSL.verify_fail_if_no_peer_cert = 2*: Si el mode verify_peer no està establert aquest mode serà ignorat. Si no es rep cap certificat per part del client la fase de handshake es donarà per acabada immediatament.

· *SSL.verify_client_once = 3*: Si el mode verify_peer no està establert aquest mode serà ignorat. El certificat del client solament es demanarà durant la fase de handshake inicial, en cap moment durant una renegociació.

depth (int)

Profunditat màxima permesa per validar la cadena de certificació.

callback

Callable que es pot utilitzar per especificar comprovacions de verificació personalitzades.

7.2.3 Estenent l'exemple

A partir de l'exemple mostrat anteriorment, afegirem el codi necessari per tal d'implementar la seguretat mostrada en aquest apartat.
Com es pot veure en el client a partir d'ara establirà les connexions utilitzant qualsevol dels protocols mencionats anteriorment, es carrega al context el certificat del client, indiquem que el mode de verificació és verify_peer i la profunditat de verificació 4 i finalment indiquem la CA amb la qual es verificaran els certificats del servidor.
En la part del servidor establim que acceptem tots els protocols de connexió, carreguem al context la cadena de certificació del servidor (servidor + CA + root), carreguem la CA amb la que verificarem els certificats dels clients i el mode de verificació és verify_peer amb profunditat 4.

```
from M2Crypto import SSL

def init_context(protocol, certfile, certkey, cafile):
    ctx = SSL.Context(protocol)
    ctx.load_cert(certfile, keyfile=certkey)
    ctx.set_verify(SSL.verify_peer, depth=4)
    ctx.load_verify_locations(cafile)
    return ctx

if __name__ == "__main__":
    certfile = "client.pem"
    certkey = "clientkey.pem"
    cafile = "rootcert.pem"

    protocol = "sslv23"

    ctx = init_context(protocol, certfile, certkey, cafile)
```

```
    SSL.Connection.clientPostConnectionCheck = None
    conn = SSL.Connection(ctx)
    conn.connect(("127.0.0.1", 2016))

    message = raw_input("Insert text to send\n")

    conn.write(message)

    conn.close()
```

Listing 7.3: Client SSL 2

```
from M2Crypto import SSL
import sys
from threading import Thread

def init_context(protocol, certfile, certkey, cafile):
    ctx = SSL.Context(protocol)
    ctx.load_cert_chain(certfile, keyfile=certkey)
    ctx.load_verify_locations(cafile)
    ctx.set_verify(SSL.verify_peer, depth=4)
    ctx.set_info_callback()
    return ctx

def get_message(ssl):
    msg = ssl.read()
    print msg
    ssl.close()

if __name__ == "__main__":
    certfile = "server.pem"
    certkey = "serverkey.pem"
    cafile = "rootcert.pem"

    addr = ("127.0.0.1", 2016)
    protocol = "sslv23"

    ctx = init_context(protocol, certfile, certkey, cafile)

    conn = SSL.Connection(ctx)

    try:
        conn.bind(addr)
    except:
        print "Bind failed"
        sys.exit(-1)
```

```
conn.listen(10)

while 1:
    ssl, addr = conn.accept()
    print "Connected with " + addr[0]
    Thread(target=get_message, args=(ssl,)).start()

conn.close
```

Listing 7.4: Servidor SSL 2

7.3 Opcions SSL i Cipher Suites

Com hem esmentat al començament d'aquest punt les aplicacions actuals ja no suporten el protocol SSLv2, però com podem observar en la nostra aplicació tant el servidor com el client poden establir una connexió utilitzant qualsevol protocol ja que no hi ha cap altra manera per ha que la nostra aplicació utilitzi els protocols que volem, SSLv3 i TLSv1. Per tant haurem de poder limitar els protocols que es poden utilitzar.
Un altre aspecte ha tenir en compte es la selecció de cipher suites. Les cipher suites són una combinació d'algorismes de baix nivell utilitzats en les connexions SSL per a l'autenticació, intercanvi de claus i encriptació del tràfic. Aquesta selecció és important perquè OpenSSL suporta alguns algoritmes que hem d'excloure per qüestions de seguretat. Altrament, algunes de les cipher suites que són segures requereixen l'aplicació de callbacks per poder ser utilitzades.
Per tant, l'objectiu d'aquest punt serà poder implementar aquests aspectes correctament estenent el nostre exemple.

7.3.1 Establint les opcions SSL

Per poder establir les opcions de SSL dins de Context utilitzarem la següent funció:

SSL.Context.set_options (self, op)

Estableix les opcions SSL.

op (int)

Opció que volem establir.
Algunes de les opcions que trobem són:

- *m2.SSL_OP_ALL*: Activa totes les correccions de bugs.
- *m2.SSL_OP_NO_SSLv2*: No es permeten connexions que utilitzin el protocol SSLv2.
- *m2.SSL_OP_NO_SSLv3*: No es permeten connexions que utilitzin el protocol SSLv3.
- *m2.SSL_OP_NO_TLSv1*: No es permeten connexions que utilitzin el protocol TLSv1.

7.3.2 Selecció de cipher suites

Una cipher suite és un conjunt d'algorismes que utilitza SSL per tal d'assegurar una connexió. Hem de proveir algorismes per a quatre funcions: signar/autenticar, intercanvi de claus, hashing criptogràfic i xifrar/desxifrar. Tot i que alguns algorismes poden servir per a diferents funcions, per exemple, RSA es pot utilitzar tant per signar com per l'intercanvi de claus.
OpenSSL implementa una gran varietat d'algorismes i cipher suites per a les connexions SSL. Per tant, quan dissenyem aplicacions segures és essencial no permetre aquells algorismes amb vulnerabilitats conegudes.
Per tal d'escollir la cipher suite utilitzarem la següent funció:

SSL.Context.set_cipher_list (self, cipher_list)

Estableix la cipher suite.

cipher_list (str)

Tipus de cipher lists que volem permetre. Cadascun dels valors es separara per ':' i s'utilitza el signe '!' per negar. Exemple: 'ALL:!ADH:!LOW:!EXP:!MD5:@STRENGTH'.

7.3.3 Ephimeral keying

En el nostre exemple els certificats utilitzen claus RSA, com RSA permet signar i xifrar, SSL ha utilitzat això per crear la clau compartida amb la qual les dades han estat encriptades. És a dir, l'intercanvi de claus s'ha fet mitjançant una clau persistent, això s'anomena static keying. Per tant, s'anomena ephimeral keying a l'intercanvi de claus mitjançant una clau temporal. Hi ha dos avantatges principals d'utilitzar ephimeral keying en comptes de static keying des del punt de vista de la seguretat. La primera és que si utilitzem claus DSA, per exemple, l'algorisme DSA permet signar, però no xifrar. Per tant, el protocol no pot dur a terme l'intercanvi de claus. Mentre que utilitzant ephimeral keying solucionaríem

aquest problema.
La segona és que al utilitzar ephimeral keiyng proporcionem confidencialitat avançada ("forward secrecy"). És a dir, si una tercera persona obté la clau privada pot desxifrar la sessió actual, però no pot desxifrar ni les sessions prèvies ni les futures sessions, ja que la clau és temporal.
Per tal d'utilitzar ephimeral keying podem fer servir Ephimeral Diffie-Hellman (EDH) o RSA. Però, RSA viola els protocols SSL/TLS, per tant el correcte és utilitzar EDH.
Per tal d'habilitar EDH utilitzarem la següent funció de l'objecte Context:

SSL.Context.set_tmp_dh (self, dhpfile)

Habilita EDH.
dhpfile (str)
Fitxer amb els paràmetres de Diffie-Hellman en format PEM.

7.3.4 Exemple final

Tant al client com al servidor utilitzem les opcions SSL d'activar totes les correccions de bugs i no permetre connexions que utilitzin el protocol SSLv2. També indiquem que acceptem totes les cipher suits (ALL) excepte les que utilitzin anonymous Diffie-Hellman (!ADH), xifratges amb claus de 64 bits o 56 bits(!LOW i !EXP), que utilitzin l'algorisme MD5 (!MD5) i tot això ordenat de major a menor resistència. I finalment al servidor passem els paràmetres de Diffie-Hellman al context per tal de poder utilitzar ephimeral keying.

```
from M2Crypto import SSL, m2

def init_context(protocol, certfile, certkey, cafile,\
 cipherlist):
    ctx = SSL.Context(protocol)
    ctx.load_cert(certfile, keyfile=certkey)
    ctx.set_verify(SSL.verify_peer, depth=4)
    ctx.load_verify_locations(cafile)
    ctx.set_options(m2.SSL_OP_ALL | m2.SSL_OP_NO_SSLv2)
    ctx.set_cipher_list(cipherlist)
    return ctx

if __name__ == "__main__":
    certfile = "clientcert.pem"
```

```
    certkey = "clientkey.pem"
    cafile = "rootcert.pem"
    cipherlist = "ALL:!ADH:!LOW:!EXP:!MD5:@STRENGTH"
    protocol = "sslv23"

    ctx = init_context(protocol, certfile, certkey,\
     cafile, cipherlist)

    SSL.Connection.clientPostConnectionCheck = None
    conn = SSL.Connection(ctx)
    conn.connect(("127.0.0.1", 2016))

    message = raw_input("Insert text to send\n")

    conn.write(message)

    conn.close()
```

Listing 7.5: Client SSL 3

```
from M2Crypto import SSL, m2
import sys
from threading import Thread

def init_context(protocol, certfile, certkey, cafile,\
 chainfile, chainkey, cipherlist, dhparam):
    ctx = SSL.Context("tlsv1")
    ctx.load_cert(certfile, keyfile=certkey)
    ctx.load_cert_chain(chainfile, keyfile=chainkey)
    ctx.load_verify_locations(cafile)
    ctx.set_verify(SSL.verify_peer, depth=4)
    ctx.set_options(m2.SSL_OP_ALL | m2.SSL_OP_NO_SSLv2)
    ctx.set_cipher_list(cipherlist)
    ctx.set_tmp_dh(dhparam)
    ctx.set_info_callback()
    return ctx

def get_message(ssl):
    msg = ssl.read()
    print msg
    ssl.close()

if __name__ == "__main__":
    certfile = "servercert.pem"
    certkey = "serverkey.pem"
    cafile = "rootcert.pem"
```

```
chainfile = "server.pem"
chainkey = "serverkey.pem"
cipherlist = "ALL:!ADH:!LOW:!EXP:!MD5:@STRENGTH"
dhparam = "dh1024.pem"
addr = ("127.0.0.1", 2016)
protocol = "sslv23"

ctx = init_context(protocol, certfile, certkey,\
 cafile, chainfile, chainkey, cipherlist, dhparam)

conn = SSL.Connection(ctx)

try:
    conn.bind(addr)
except:
    print "Bind failed"
    sys.exit(-1)

conn.listen(10)

while 1:
    ssl, addr = conn.accept()
    print "Connected with " + addr[0]
    Thread(target=get_message, args=(ssl,)).start()

conn.close
```

Listing 7.6: Servidor SSL 3

Capítol 8

HTTPS

Un dels principals protocols més utilitzats avui en dia és Hypertext Transfer Protocol (HTTP) destinat a l'enviament d'hipertext, principalment utilitzat a Internet. Aquest protocol no estableix cap tipus de xifratge i les dades s'envien en text pla, per tant, no compleix amb els requisits de confidencialitat, integritat, autenticació i no repudi de les dades. D'aquí sorgeix Hypertext Transfer Protocol Secure (HTTPS) [8.1] basat en el protocol anterior on mitjançant SSL crea un canal xifrat per tal de poder transferir hipertext. Així les dades viatgen xifrades i solament poden ser desxifrades pel client i el servidor. Tant en la part de client com de servidor utilitzarem la jerarquia de certificats creada en el capítol 6, comes veu a la Fig. 6.4.

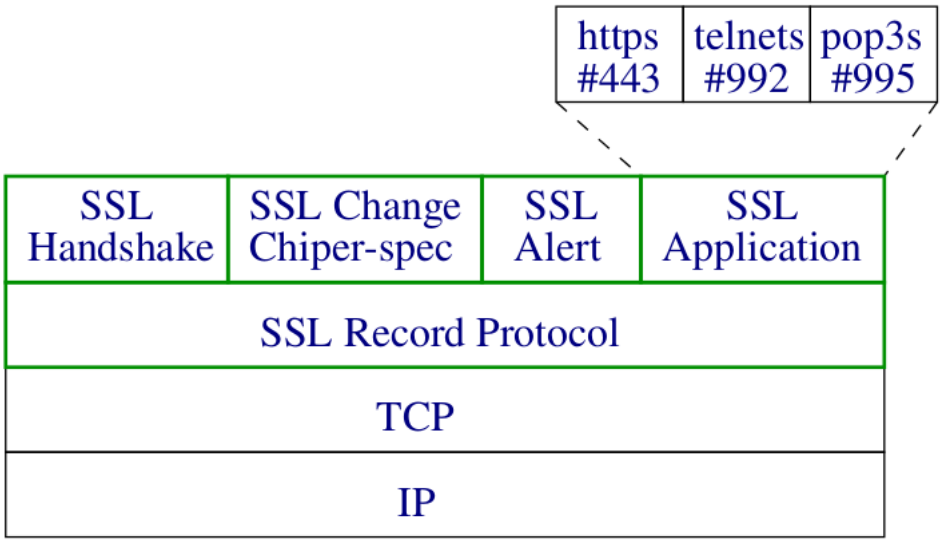

Figura 8.1: Arquitectura SSL

8.1 Servidor HTTPS

L'objectiu del servidor HTTPS és establir un canal xifrat mitjançant SSL per tal d'atendre les peticions HTTP en un port concret. El mòdul SSL de la llibreria M2Crypto ens proporciona la classe SSLServer, que estén SocketServer.BaseServer, la qual arrenca un servidor SSL directament i atén les peticions que rep a partir del handler que s'indica a través del constructor. El constructor és el següent:

SSLServer (self, server_address, RequestHandlerClass, ssl_context, bind_and_activate=True)

server_addres (str, int)

Tupla amb l'adreça del servidor i el port pel qual volem establir la connexió.

RequestHandlerClass

Classe que utilitzarem com a handler per atendre les peticions que es rebin.

ssl_context

Context SSL per tal d'establir la connexió SSL amb el client.

bind_and_activate (bool)

Booleà per establir si volem que el servidor estigui actiu i escoltant.

En el nostre cas utilitzarem la classe ThreadingSSLServer, que estén SSLServer, i utilitza el mateix constructor ja que la funcionalitat és la mateixa tot i que en aquest cas crea un thread per atendre cada petició.
Per saber com crear el context SSL consultar l'apartat 7, SSL.
Com a handler utilitzarem la classe SimpleHTTPRequestHandler, del mòdul SimpleHTTPServer [13], el qual rep peticions sobre un directori o fitxer i retorna el contingut d'aquest directori o fitxer, molt similar al funcionament del servidor HTTP Apache.

```
from SimpleHTTPServer import SimpleHTTPRequestHandler

from M2Crypto import SSL
from M2Crypto.SSL.SSLServer import ThreadingSSLServer

class HTTPS_Server(ThreadingSSLServer):
    def __init__(self, server_addr, handler, ssl_ctx):
        ThreadingSSLServer.__init__(self, server_addr,\
```

```
            handler, ssl_ctx)
        self.server_name = server_addr[0]
        self.server_port = server_addr[1]

    def finish(self):
        self.request.set_shutdown(
        SSL.SSL_RECEIVED_SHUTDOWN | SSL.SSL_SENT_SHUTDOWN)
        self.request.close()


def init_context(protocol, certfile, certkey,\
 cafile, chainfile, chainkey, verify_depth=4):
    ctx=SSL.Context(protocol)
    ctx.load_cert(certfile, keyfile=certkey)
    ctx.load_cert_chain(chainfile, keyfile=chainkey)
    ctx.load_client_ca(cafile)
    if ctx.load_verify_locations(cafile) != 1:
        raise Exception('CA certificates not loaded')
    ctx.set_verify(SSL.verify_peer |\
     SSL.verify_fail_if_no_peer_cert, verify_depth)
    ctx.set_info_callback()
    return ctx

if __name__ == '__main__':
    certfile = "servercert.pem"
    certkey = "serverkey.pem"
    cafile = "rootcert.pem"
    chainfile = "server.pem"
    chainkey = "serverkey.pem"
    ctx = init_context('sslv23', certfile, certkey,\
    cafile, chainfile, chainkey)
    httpsd = HTTPS_Server(('localhost', 9443),\
     SimpleHTTPRequestHandler, ctx)
    httpsd.serve_forever()
```

Listing 8.1: Servidor HTTPS

8.2 Client HTTPS

L'objectiu del client HTTPS és establir una connexió SSL amb el servidor per tal de poder enviar peticions HTTP en un port determinat.

La llibreria M2Crypto ens proporciona el mòdul httpslib que és una extensió de httplib. Dins del d'aquest mòdul trobem la classe HTTPSConnection, que estén httplib.HTTPConnection, la qual s'encarrega d'establir la connexió a partir de

l'adreça, el port i el context SSL. El seu constructor és el següent:

HTTPSConnection (self, host, port, ssl_context=None)

host (str)

Adreça del servidor a connectar-se.

port (int)

Port en el qual s'establirà la connexió.

ssl_context

Context SSL per tal d'establir la connexió SSL amb el client.

Les principals funcions que utilitzarem són:

connect()

Estableix la connexió amb el servidor.

putrequest(request, request-uri)

Estableix la petició.

request (str)

Petició a enviar al servidor. Exemple: 'GET', 'POST', 'PUT'...

request-uri (str)

URI de la petició.

putheader(header, argument)

Estableix la capçalera de la petició.

header(str)

Capçalera de la petició.

argument(str[])

Argument/s de la capçalera.

endheaders()

Indica el final de les capçaleres.

send(data)

Envia el cos del missatge.

data (str)

Cos del missatge.

```
from M2Crypto import httpslib, SSL, m2

def init_context(protocol, certfile, certkey, cafile):
    ctx = SSL.Context(protocol)
    ctx.load_cert(certfile, keyfile=certkey)
    ctx.set_verify(SSL.verify_peer, depth=4)
    ctx.set_options(m2.SSL_OP_ALL | m2.SSL_OP_NO_SSLv2)
    if ctx.load_verify_locations(cafile) != 1:
        raise Exception('CA certificates not loaded')
    return ctx


if __name__ == '__main__':

    certfile = "clientcert.pem"
    certkey = "clientkey.pem"
    cafile = "rootcert.pem"

    ctx = init_context("sslv23", certfile, certkey, cafile)

    SSL.Connection.clientPostConnectionCheck = None

    httpsconn = httpslib.HTTPSConnection('localhost',\
     9443, ssl_context=ctx)
    httpsconn.connect()
    httpsconn.putrequest('GET', '/index.html')
    httpsconn.putheader('Accept', 'text/html')
    httpsconn.putheader('Connection:', 'keep-alive, close')
    httpsconn.endheaders()
    httpsconn.send("data")

    resp = httpsconn.getresponse()
    print resp.read()

    httpsconn.close()
```

Listing 8.2: Client HTTPS

Capítol 9

SMIME

Secure Multipurpose Internet Mail Extensions (SMIME) [9.1] és un conjunt d'estàndards emprat per a criptografia de clau pública y signat de correu electrònic. Empra una Infraestructura de clau pública (PKI) per tal de dotar al correu electrònic de confidencialitat, autenticitat i no repudi, també empra les claus públiques dels destinataris per xifrar una clau de sessió.
La primera versió va ser desenvolupada per RSA Security l'any 1995, actualment es troba en la versió 3. La versió 3 consta de les següents 5 parts:

- Sintaxi de missatges criptogràfics (CMS). Compatible amb PKCS#7. [7]
- Algorismes per a CMS. [6]
- Especificació de missatges S/MIME v3.1. [15]
- Gestió de certificats S/MIME v3.1. [14]
- Intercanvi de claus Diffie-Hellman. [16]

Utilitza certificats basats en x509, els algorismes de xifrat 3DES-CBC i RC2-CBC, l'algorisme de signatura DH amb RSA o DSS, l'algorisme de hash SHA-1 i com a contenidor pkcs#7.

Els tipus de missatges que podem generar són:

- Signed: missatge + una o més signatures.
- Enveloped: missatge xifrat + claus de sessió xifrades.
- Digested: missatge + hash. Sol emprar-se com a input d'Enveloped.
- Encrypted: missatge xifrat únicament. Les claus s'han de negociar a banda.

- Authenticated: missatge + MAC + claus xifrades.

La llibreria ens proporciona el mòdul SMIME i les següents funcions per tal d'obtenir el missatge SMIME o el contenidor en format PKCS#7:

load_pkcs7(p7file)

Retorna un objecte de tipus PKCS7 que contindrà el contenidor en format PKCS#7.

p7file (str)

Fitxer que conté el contenidor en format PKCS#7 del missatge.

load_pkcs7_bio(p7bio)

Retorna un objecte de tipus PKCS7 que contindrà el contenidor en format PKCS#7.

p7bio

Objecte de tipus BIO que conté el contenidor en format PKCS#7 del missatge.

smime_load_pkcs7(p7file)

Retorna una tupla amb un objecte de tipus PKCS7 que contindrà el contenidor en format PKCS#7 i les dades del missatge.

p7file (str)

Fitxer que conté el missatge en format SMIME.

smime_load_pkcs7_bio(p7bio)

Retorna un objecte de tipus PKCS7 que contindrà el contenidor en format PKCS#7.

p7bio

Objecte de tipus BIO que conté el missatge en format SMIME.

text_crlf(text)

Retorna el contingut del missatge formatat.

text (str)

Contingut del missatge.

text_crlf_bio(text)

Retorna un objecte de tipus BIO amb el contingut del missatge formatat.

text (str)

Contingut del missatge.

La classe PKCS7 ens proporciona els següents mètodes:

type(self, text_name=0)

Retorna el tipus del contenidor PKCS#7.

text_name (int)

Si val 0 retorna el tipus en format enter i si val 1 retorna el tipus en format string. Els diferents tipus són:

- PKCS7_SIGNED = 22
- PKCS7_ENVELOPED = 23
- PKCS7_SIGNED_ENVELOPED = 24

get0_signers(self, certs, flags=0)

Retorna un objecte X509_Stack amb els certificats que conté el pkcs7.

certs

Objecte X509_Stack.

flags

Flags a utilitzar.

write(self, bio)

Afegeix el contingut de bio dins del contenidor PKCS#7.

bio

Objecte de tipus BIO amb les dades a escriure.

La classe SMIME ens proporciona els següents mètodes:

load_key(self, keyfile, certfile=None, callback

Carrega la clau privada dins de l'objecte SMIME.

keyfile (str)

Fitxer que conté la clau privada del certificat.

certfile (str)

Fitxer que conté el certificat.

callback

Objecte de tipus Callback utilitzat per obtenir la contrasenya de la clau si fos necessària. Per defecte la demana per consola.

load_key(self, keybio, certbio=None, callback

Carrega la clau privada dins de l'objecte SMIME.

keyfile (str)

Objecte de tipus BIO que conté la clau privada del certificat.

certfile (str)

Objecte de tipus BIO que conté el certificat.

callback

Objecte de tipus Callback utilitzat per obtenir la contrasenya de la clau si fos necessària. Per defecte la demana per consola.

set_cipher(self, cipher)

Selecciona el tipus de xifratge que s'utilitzarà.

cipher

Objecte de tipus Cipher que contindrà el tipus de xifratge a emprar.

unset_cipher(self)

Elimina el tipus de xifratge seleccionat.

set_x509_stack(self, stack)

Afegeix la pila de certificats del recipient dins de l'objecte SMIME.

stack

Objecte de tipus X509_Stack que conté el certificat del recipient.

unset_x509_stack(self)

Elimina la pila de certificats del recipient.

set_x509_store(self, store)

Afegeix la cadena de certificació dins de l'objecte SMIME.

stack

Objecte de tipus X509_Store que conté la cadena de certificació.

unset_x509_store(self)

Elimina la cadena de certificació.

encrypt(self, data_bio, flags=0)

Retorna un objecte PKCS7 que conté les dades xifrades.

data_bio

Objecte de tipus BIO que conté les dades a xifrar.

flags (int)

Flag a utilitzar.

decrypt(self, pkcs7, flags=0)

Retorna el missatge del contenidor PKCS#7 desxifrat.

pkcs7

Objecte de tipus PKCS7 que conté el missatge xifrat.

flags (int)

Flag a utilitzar.

sign(self, data_bio, flags=0)

Retorna un objecte PKCS7 que conté les dades signades.

data_bio

Objecte de tipus BIO que conté les dades a signar.

flags

Flag a utilitzar.

verify(self, pkcs7, data_bio=None, flags=0)

Verifica les dades dins del contenidor PKCS#7.

pkcs7

Objecte de tipus PKCS7 que conté les dades signades

data_bio

Objecte de tipus BIO amb el missatge rebut si es vol validar també a partir d'aquest

flags (int)

Flag a utilitzar.

write(self, out_bio, pkcs7, data_bio=None, flags=0)

Dona format al missatge SMIME i escriu les dades dins del contingut d'aquest.

out_bio

Objecte de tipus BIO on s'escriurà el missatge en format SMIME.

pkcs7

Objecte de tipus PKCS7 amb el contenidor PKCS#7

data_bio

Objecte de tipus BIO amb les dades del contingut del missatge.

flags

Flag a utilitzar.

Els flags que podem utilitzar són:

- *SMIME.PKCS7_TEXT = 1*: s'afegeixen capçaleres MIME (text/plain) a l'inici.
- *SMIME.PKCS7_NOCERTS = 2*: no s'afegeixen certificats addicionals en el pkcs7.
- *SMIME.PKCS7_NOSIGS = 4*: no es verifica la signatura de l'objecte pkcs7, però si els certificats.
- *SMIME.PKCS7_NOCHAIN = 8*: es verifica la cadena de certificació contra l'store (X509_Store), obviant els certificats de l'estructura PKCS7.
- *SMIME.PKCS7_NOINTERN = 16*: no s'empren els certificats inclosos a l'objecte pkcs7 per verificar, si no els continguts a l'stack (X509_Stack).
- *SMIME.PKCS7_NOVERIFY = 32*: no es verifica res.
- *SMIME.PKCS7_DETACHED = 64*: no s'inclou les dades signades.
- *SMIME.PKCS7_BINARY = 128*: no codifica les dades.

```
MIME-Version: 1.0
Content-Type: multipart/signed; protocol="application/x-pkcs7-signature"; micalg="sha-256"; boundary="----CC40C287EB30830E48BE8FD3FFFF3B33"

This is an S/MIME signed message

------CC40C287EB30830E48BE8FD3FFFF3B33
Content-Type: text/plain

Missatge signat

------CC40C287EB30830E48BE8FD3FFFF3B33
Content-Type: application/x-pkcs7-signature; name="smime.p7s"
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename="smime.p7s"


N40DesJACBFYp2RpsYSrYKr3r0un5YAXzqIBX31Aw4s6U7zlDPBz9nv87r6/kugl
MYICEzCCAg8CAQEwgYEwdDELMAkGA1UEBhMCRVMxDzANBgNVBAcMBkxsZWlkYTEc
MBoGA1UECgwTRGVmYXVsdCBDb21wYW55IEx0ZDEQMA4GA1UEAwwHcm9vdF9DQTEk
MCIGCSqGSIb3DQEJARYVYWNtMjRAYWx1bW5lcy51ZGwuY2F0AgkArar1INUvd78w
DQYJYIZIAWUDBAIBBQCggeQwGAYJKoZIhvcNAQkDMQsGCSqGSIb3DQEHATAcBgkq
hkiG9w0BCQUxDxcNMTYwNTI1MTYwNjUzWjAvBgkqhkiG9w0BCQQxIgQgCINNcKoX
P4iwJfJYQTZwMf09XoPBLGhpTLBOBd4AZPgweQYJKoZIhvcNAQkPMWwwajALBglg
hkgBZQMEASowCwYJYIZIAWUDBAEWMAsGCWCGSAFlAwQBAjAKBggqhkiG9w0DBzAO
BggqhkiG9w0DAgICAIAwDQYIKoZIhvcNAwICAUAwBwYFKw4DAgcwDQYIKoZIhvcN
AwICASgwDQYJKoZIhvcNAQEBBQAEgYBahx4zvAezgVLHReURg0rptlFpOLGYLuob
L/fSo5ybz8+g1hUll2QxQx+lyCmPTGXkmBIZPXya/x2dmQxw/h4nJ/nlIf8ABAhm
/5Mbv/XTG/+AXsFZImBz+ossfePtN9HROViHr302RZwfh0FlcSCVImc7TPBGpdhz
DtaQQ9kbXA==

------CC40C287EB30830E48BE8FD3FFFF3B33--
```

Figura 9.1: Exemple de missatge en format SMIME

9.1 Exemple SMIME

En aquest exemple utilitzarem la jerarquia de certificats creada en el capítol 6, comes veu a la Fig. 6.4.

```
import smtplib
from M2Crypto import BIO, SMIME

buf = BIO.MemoryBuffer(SMIME.text_crlf("Missatge signat"))

s = SMIME.SMIME()
s.load_key('clientkey.pem', 'clientcert.pem')
p7 = s.sign(buf, SMIME.PKCS7_DETACHED)

buf = BIO.MemoryBuffer(SMIME.text_crlf("Missatge signat"))

out = BIO.MemoryBuffer()
out.write('From: correu@servidor\n')
out.write('To: correu@servidor\n')
out.write('Subject: Missatge signat\n')
s.write(out, p7, buf)

smtp = smtplib.SMTP('servidor SMTP')
smtp.sendmail('from', 'to', out.read())
```

Listing 9.1: Signar missatge SMIME i enviar

Per tal de comprovar si el nostre missatge ha estat signat correctament podem utilitzar el client de correu Thunderbird [3]. Primerament hem d'afegir el certificat root a la llista de CA de Thunderbird, per fer-ho hem d'anar a preferències, a l'apartat Avançat i a la pestanya Certificats com es veu a la Fig. 9.2. Després ja podem rebre correus signats per certificats emesos per root i si tot es correcte a l'obrir el missatge es veurà la icona d'un missatge segellat on al clica sortirà un missatge conforme el missatge s'ha verificat correctament, com es veu a la Fig. 9.3.

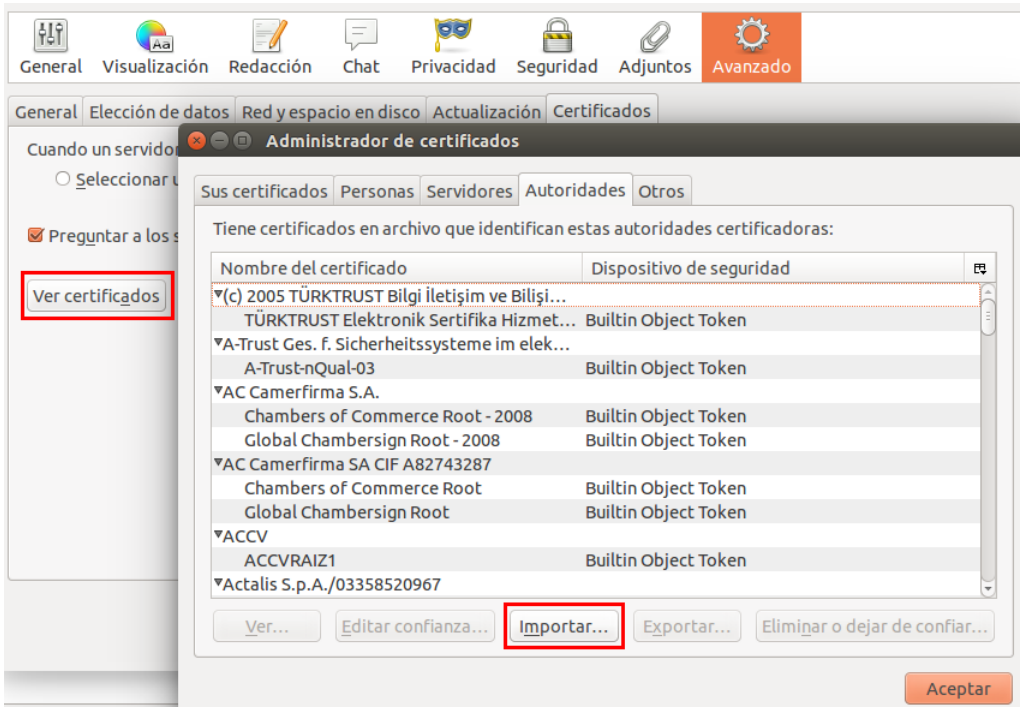

Figura 9.2: Afegir certificat a la llista de CA's de Thunderbird

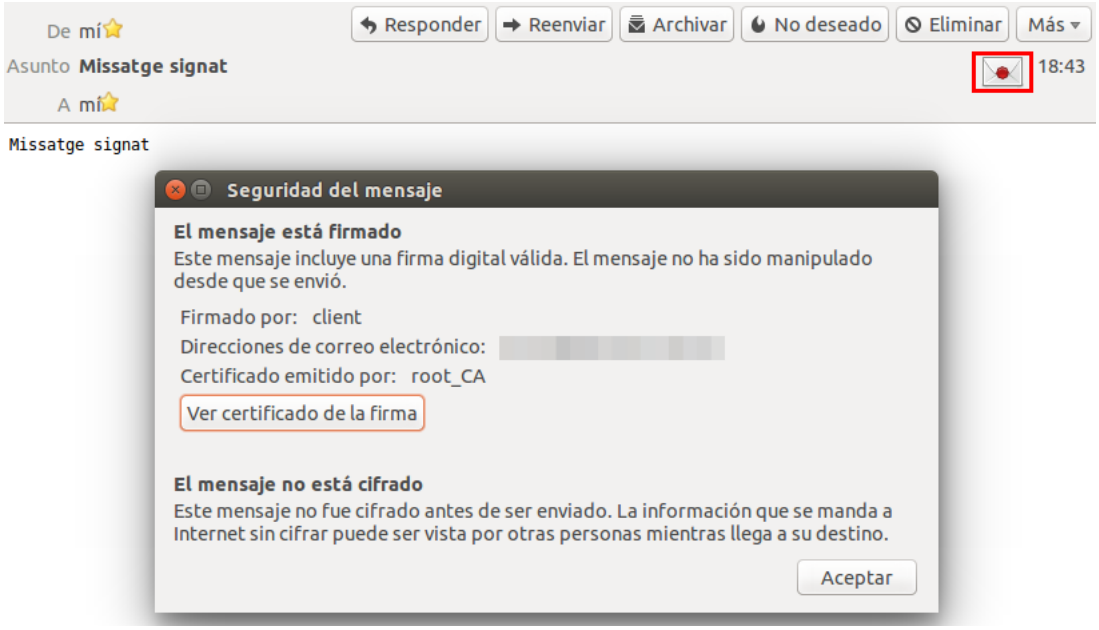

Figura 9.3: Missatge verificat correctament

Capítol 10

Conclusions

En conclusió, primerament hem vist com implementar en python les aplicacions més simples d'OpenSSL, el xifrat simètric i el xifrat de clau pública. Per tal de poder intercanviar claus hem vist com implementar l'intercanvi de claus Diffie-Hellman. Un cop hem après a implementar el xifrat de clau pública hem vist com crear certificats digitals per tal de poder validar la nostra identitat. Per tal de poder utilitzar aquestes aplicacions en l'àmbit de les comunicacions hem implementat els protocols SSL i TLS i l'aplicació d'aquests protocols en HTTPS. Finalment, hem implementat les aplicacions d'SMIME per tal de poder dur a terme l'ús de missatgeria segura.
Tot això complint el nostre objectiu de facilitar la comprensió de les aplicacions d'OpenSSL gràcies a la utilització d'objectes i evitant haver d'administrar l'espai en memòria ja que amb l'ús del wrapper d'OpenSSL M2Crypto tot això és aliè a l'usuari.

Bibliografia

[1] Matej Cepl. M2crypto home page. https://gitlab.com/m2crypto/m2crypto.

[2] César Fernández. Transparències seguretat d'aplicacions i comunicacions, 2015.

[3] Mozilla Foundation. Thunderbird homepage. https://www.mozilla.org/es-ES/thunderbird/.

[4] M. Bellare H. Krawczyk and R. Canetti. Hmac: Keyed-hashing for message authentication. https://tools.ietf.org/html/rfc2104, 1997.

[5] D. Cooper S. Santesson S. Farrell S. Boeyen R. Housley and W. Polk. Internet x.509 public key infrastructure certificate and certificate revocation list (crl). http://tools.ietf.org/rfc/rfc5280, 2008.

[6] R. Housley. Cryptographic message syntax (cms) algorithms. https://tools.ietf.org/html/rfc3370, 2002.

[7] R. Housley. Cryptographic message syntax (cms). https://tools.ietf.org/html/rfc3852, 2014.

[8] J. Jonsson and B. Kaliski. Public-key cryptography standards (pkcs) #1: Rsa cryptography specifications version 2.1. https://tools.ietf.org/html/rfc3447, 2003.

[9] B. Kaliski. Pkcs #5: Password-based cryptography specification version 2.0. https://tools.ietf.org/html/rfc2898, 2000.

[10] S. Legg. Generic string encoding rules (gser) for asn.1 types. https://tools.ietf.org/html/rfc3641, 2003.

[11] Pravir Chandra Matt Messier and John Viega. *Network Security with OpenSSL*. O'Reilly, 2002.

[12] National Institute of Standards and Technology. Digital signature standard (dss). http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf, 2013.

[13] Simplehttpserver python documentation. https://docs.python.org/2/library/simplehttpserver.html.

[14] B. Ramsdell. Secure/multipurpose internet mail extensions (s/mime) version 3.1 certificate handling. `https://tools.ietf.org/html/rfc3850`, 2004.

[15] B. Ramsdell. Secure/multipurpose internet mail extensions (s/mime) version 3.1 message specification. `https://tools.ietf.org/html/rfc3851`, 2004.

[16] E. Rescorla. Diffie-hellman key agreement method. `https://tools.ietf.org/html/rfc2631`, 1999.

[17] William Stallings. *Cryptography and Network Security Principles and Practices*. Prentice Hall, 2005.

[18] H. Toivonen. M2crypto 0.20 documentation. `http://www.heikkitoivonen.net/m2crypto/api/M2Crypto-module.html`.

[19] X.500 standard. `http://www.x500standard.com/`.

Acrònims

ASN.1 Abstract Syntax Notation One. 45

CA Autoritat Certificadora. 37

CMS Sintaxi de missatges criptogràfics. 81

CRL Certificate Revocation List. 46

DSA Digital Signature Algorithm. 29

EDH Ephimeral Diffie-Hellman. 71

HMAC Hash-based Message Authentication Code. 18

HTTP Hypertext Transfer Protocol. 75

HTTPS Hypertext Transfer Protocol Secure. 75

MAC Codis d'Autentificació de Missatges. 9

PBKDF Password-Based Key Derivation Function. 15

PKI Infraestructura de clau pública. 81

RSA Rivest Shamir Adleman. 21

SMIME Secure Multipurpose Internet Mail Extensions. 81

SSL Secure Socket Layer. 9, 61

TLS Transport Socket Layer. 9, 61

Codi font